Imperial College London
Department of Computing

Dynamic Data Placement and Discovery in Wide-Area Networks

Nicholas Ball

October 1, 2013

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Abstract

The workloads of online services and applications such as social networks, sensor data platforms and web search engines have become increasingly global and dynamic, setting new challenges to providing users with low latency access to data. To achieve this, these services typically leverage a multi-site wide-area networked infrastructure. Data access latency in such an infrastructure depends on the network paths between users and data, which is determined by the *data placement* and *discovery strategies*. Current strategies are static, which offer low latencies upon deployment but worse performance under a dynamic workload.

We propose *dynamic* data placement and discovery strategies for wide-area networked infrastructures, which adapt to the data access workload. We achieve this with *data activity correlation* (DAC), an application-agnostic approach for determining the correlations between data items based on access pattern similarities. By dynamically clustering data according to DAC, network traffic in clusters is kept local. We utilise DAC as a key component in reducing access latencies for two application scenarios, emphasising different aspects of the problem:

The first scenario assumes the fixed placement of data at sites, and thus focusses on data discovery. This is the case for a global sensor discovery platform, which aims to provide low latency discovery of sensor metadata. We present a self-organising hierarchical infrastructure consisting of multiple DAC clusters, maintained with an online and distributed *split-and-merge* algorithm. This reduces the number of sites visited, and thus latency, during discovery for a variety of workloads.

The second scenario focusses on data placement. This is the case for global online services that leverage a multi-data centre deployment to provide users with low latency access to data. We present a geo-dynamic partitioning middleware, which maintains DAC clusters with an online *elastic partition* algorithm. It supports the geo-aware placement of partitions across data centres according to the workload. This provides globally distributed users with low latency access to data for static and dynamic workloads.

Dedicated to my wife and family.

Acknowledgements

I would like to begin my acknowledgements by thanking my supervisor, Peter Pietzuch, for providing me with the incredible opportunity to pursue a doctorate degree and achieve a major milestone in my life. His guidance, encouragement and support is what has made this thesis possible, and for this I will always be grateful. He has taught me how to think critically and ask the right questions, invaluable skills for the future. Thank you for all this.

During my years at Imperial College London I have had the pleasure of being part of the Large-Scale Distributed Systems (LSDS) group, forming many friendships: Eva, Ioannis, Thom, Raul, Alex, Lukas and Luo, thank you for making my time at Imperial enjoyable. Our conversations in the office and pub have been truly stimulating. A special thank you to Andreas Pambouris for taking the loneliness out of a Ph.D., and for his support and friendship throughout.

I would like to thank my family, for their encouragement and motivation throughout these years. My parents for always reminding me of the bigger picture in life, a balance which has been critical for me. My brothers and sisters for being both understanding and supportive, always wanting the best for me.

Finally I would like to express my deepest gratitude to my wife, Nadine Hawa. For whom has given me the unconditional love and support I needed throughout this doctorate. Her patience and stubbornness to stand by me in both good and bad, has been crucial to this thesis and for that I am eternally grateful. Thank you for persistently managing to cheer me up and thank you for always believing in me. I dedicate this thesis to her.

— London, September 2013

Declaration

This thesis presents my work in the Department of Computing at Imperial College London between October 2009 and September 2013.

I declare that the work presented throughout this thesis is my own, except where otherwise acknowledged.

Copyright Declaration

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

For globally distributed online services and applications such as social networks,[1][2] sensor data platforms [NLZ07, CGN05, SPL+04, CSWH01] and web search engines[3] to provide users with rich and personalised functionality, users must access and manipulate stored data. The networked infrastructures for these services and applications vary greatly depending on their purpose and popularity. For example, they can span from: millions of unreliable and poorly resourced nodes, as in the case of sensor data platforms; to half-a-dozen highly available and well-provisioned data centres (DCs), each comprising of hundreds or thousands of machines, as found in web search engines. In this thesis, we investigate techniques for reducing network latencies for users accessing data of global online services and applications. We focus on *multi-site* networked infrastructures interconnected by a wide-area network (WAN). We describe the problem using two application scenarios, each with different network properties and data management restrictions:

Following the vision of the "Internet of Things" (IoT) [Ash09], the first scenario aims to interconnect the world's sensors embedded throughout everyday physical objects into a single Internet infrastructure. The motivation for this is to leverage the connectivity of a large number of real-time data streams to develop and operate smart homes, offices and cities [MMR+08], early warning disaster detection systems [LGBS09] and advanced healthcare applications [MFJWM04]. As a first step towards this vision, a platform is required that can *discover* relevant sensor data sources given a user query. Such a platform must provide globally distributed users with low latency access to potentially millions of data sources.

The second scenario is a global online service, for example, an online social network (OSN) such as Facebook,[1] a micro-blogging service such as Twitter[2] or a web search engine such as Google.[3] These services provide users with a rich and interactive application experience by accessing and manipulating application data stored on back-end distributed data stores. These data stores are increasingly deployed across multiple DCs, consisting of either dedicated enterprise DCs or a global DC cloud infrastructure. Online services aim to improve

[1] Facebook, `http://www.facebook.com/`, last accessed: 19/3/2013
[2] Twitter, `http://www.twitter.com/`, last accessed: 19/3/2013
[3] Google, `http://www.google.com/`, last accessed: 19/3/2013

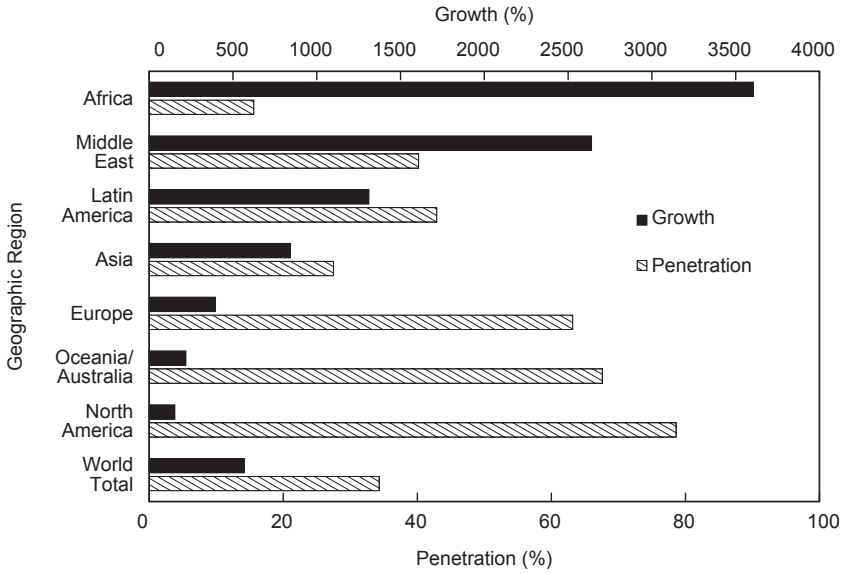


Figure 1.1: Internet usage growth (2000-2012) and penetration (Q2 2012) across the world[4]

the user experience of applications by increasing the responsiveness of their user interfaces (UI) and achieve this by providing low latency access to application data.

The trend in global Internet connectivity has brought the above online services and applications to the pockets and doorsteps of over two billion users worldwide.[4] Figure 1.1 illustrates a global increase in Internet adoption of over 560% between 2000 and 2012, with the majority of growth found in previously under-provisioned regions such as Africa, the Middle East and Latin America. Internet penetration rates—defined as a percentage of the population—remain low in these regions (15.6%, 40.2% and 42.9%, respectively) compared to North America (78.6%), indicating further opportunities for growth in the future.

This rapid and geographically dispersed growth in Internet connectivity has changed the landscape of workloads for multi-site wide-area networked infrastructures, with data access requests being received from an increasingly *global* user base. The added geographical distance between users and infrastructure has increased the length of the network path used to communicate over the Internet [KPL+09, SPK02, LC13, SWZ07]. Packets are being sent further distances and through more routers, reducing the quality-of-service (QoS) provided and resulting in increased delays, jitter and packet loss. Many users are therefore experiencing high network latencies and thus poor response times.

Workloads have also become more *dynamic*, with the interactions between users and data determined by an unprecedented number of variables. Some of these factors lead to: frequent shifts in content popularity and user interests; evolution of social connections; and changes in access and privacy restrictions, application features and application logic [BSW12, YL11, KLPM10, ZSW+12, VMCG09, KKNG12]. The network path lengths of such interactions are also influenced by factors such as the mobility habits of users and the devices used, determining the communication medium [SNLM11, NSMP11, CCLS11].

[4]Internet World Stats 2012, `http://www.internetworldstats.com/stats.htm`, last accessed: 19/3/2013

An effective solution to shortening the network path length between user and data in a multi-site wide-area networked infrastructure is to place the data at a site that is "closest" to the user. Proximity can be defined in terms of latency or more commonly approximated by the number of network routing hops, geographic distance [KPL+09] or the overlap between IP address prefixes of two hosts [FVFB05]. This is achieved through the adoption of an appropriate *data placement strategy*—the policy for the placement of data across a given multi-site infrastructure. For the application scenario of a global online service, the strategy defines the placement of application data across multiple geographically distributed DCs.

An alternative solution is to ensure that the length of the combined network paths between users and data is reduced. This is achieved through the adoption of an appropriate *data discovery strategy*—the application-level routing policy of requests over a multi-site networked infrastructure based on the organisation of sites and the application-level links between them. For the application scenario of a global sensor discovery platform, the strategy controls the routing of user requests across a network of thousands of sensor data sources.

Under a global and dynamic workload, it is the effectiveness of both placement and discovery strategies combined that dictates the data access latencies. To explore different strategies, we use the two application scenarios previously described to focus on data placement and data discovery, independently.

In the remaining sections of this chapter, we discuss the two application scenarios in further detail (Section 1.1), outlining the motivation for this thesis and the role of data placement and discovery for each. We then briefly discuss existing strategies in Section 1.2, followed by an overview of the work presented in this thesis and how it differs from existing solutions (Section 1.3). We end the chapter with the contributions of the thesis in Section 1.4, followed by a summary of each chapter in Section 1.5.

1.1 Application scenarios

To investigate the reduction of network latencies in multi-site wide-area networked infrastructures, we explore two application scenarios with different network properties and data management restrictions. In the first scenario, we focus on data discovery, investigating strategies to interconnect a large number of sensor data sources. In the second scenario, we focus on data placement, investigating strategies to place application data across multiple DCs.

1.1.1 Global sensor discovery

An increasing number of sensors are being deployed into the world in the hope to solve some of today's most pressing issues with the data gathered. Air pollution monitoring [Pol07, ASC+10], physiological monitoring [MFJWM04], environmental disaster warning systems [HB09] and coral reef ecosystem monitoring [PABB05] are just a few examples of deployed sensor

networks. Besides these existing networks, we are witnessing a mushrooming of individual sources of sensor data on the Internet. For example, modern smartphones are equipped with a range of sensors that can provide a stream of sensed data [BEH06]. Wireless sensor networks are also being deployed in both rural and urban environments with an up-link to the Internet, making their data continuously available [MMR+08, MMG+08, LGBS09, RB07]. Finally, Internet websites and online services themselves provide streaming data sources in the form of RSS feeds and asynchronous notifications [RMP07, AMO13].

Although individual sensor readings provide limited information, when *aggregated* across thousands of data sources, more insightful conclusions can be derived. This is the vision for the "Internet of Things" (IoT) [Ash09], a paradigm which calls for all sensors and actuators embedded seamlessly throughout our environment to collaborate and communicate amongst themselves, users and autonomous applications. The realisation of the IoT paradigm requires the convergence of three areas outlined by Atzori et al. [AIM10], which include (1) *things oriented*—the development of low-energy consumption, smaller and lighter sensors, tags and actuators such as Radio-Frequency IDentification (RFID) tags [FW99], (2) *internet oriented*—the middleware for interconnecting "things" in a scalable and efficient global infrastructure and (3) *semantic oriented*—the interoperability of various identification, data representation and storage standards of heterogeneous "things" to transform raw low-level sensor data into actionable knowledge.

The convergence of these three areas combined with the millions of hardware and software data sources available to us can support an almost infinite number of applications. These range from smart homes, offices and cities [MMR+08],[5] early warning disaster detection systems [LGBS09] and advanced healthcare applications [MFJWM04]. More interestingly is the prospect of applications utilising a *combination* of data sources. For example, medical software deployed in a third-world country that tracks the spread of diseases and illnesses such as tuberculosis and malaria [LWQ10] can be augmented with an early-warning humanitarian aid system. Such a system could inform the right organisations *when* a particular region will require aid, *what* variety and amount of aid, *where* it should be deployed and exactly *how* this plan should be executed.

All of the applications above rely on means to discover relevant sources of sensor data according to application requirements—a facet of the *internet oriented* area. For example, our humanitarian aid system may require data from temperature, humidity and rainfall sensors to calculate the likelihood of a malaria outbreak. However before any data streams can be received, the relevant sensors must be found. Sensor discovery is achieved by executing user requests against the metadata of sensor data sources, with the accuracy of results dependant on metadata freshness. Performing discovery on outdated metadata—introduced by caching or replication—could lead to outdated and thus inaccurate results.

For the advancement of the IoT vision, we investigate the development of a global sensor

[5] Smart Roads - Wireless Sensor Networks for Smart Infrastructures, `http://www.libelium.com/smart_roads_wsn_smart_infrastructures/`, last accessed: 10/8/2013

discovery platform comprising of an infrastructure of sensor data sources built over the Internet. To provide accurate results, we avoid caching or replication and assume the fixed placement of metadata, thus focussing on data discovery. Thousands of globally distributed data sources or nodes must be organised into a network that ensures fast and efficient search of fresh metadata. The main contributor to the latency during data source discovery is the *total* network path length between users and the metadata returned. This length is determined by the sum of paths between nodes (where the number of paths is defined as *hops*), when routing a request through the network.

Given a set of globally distributed sensor data sources and a global user base requesting sensor metadata, a key challenge arises regarding *data discovery*: how should the appropriate organisation of data sources into a network and the forwarding of requests upon this structure be performed to ensure requests can be handled with low hop count, and therefore low network latency. While both placement and discovery strategies combined determine the latency of a global sensor discovery platform, we assume the fixed placement of metadata across data sources. We therefore focus on data discovery for this application scenario.

1.1.2 Global online services

Global online services such as Facebook, Twitter and Google strive to reduce user-perceived latencies. Latency is a major factor in user experience and has been shown to affect both user engagement and satisfaction. Furthermore, engagement and satisfaction directly influences service revenue,[6] marking latency as a high priority performance metric to optimise. Amazon Store have stated that an added tenth of a second to user response time costs the company 1% in sales.[7] Google have concluded that half a second increase in response time for their users causes traffic to drop by one fifth.[8,9] An increased delay of two seconds for Microsoft Bing[10] resulted in 1.8% reduction in queries per user and a 4.3% reduction in revenue per user.[11] When applied to the large volume of traffic handled by such services, any latency improvement has a major impact on revenue.

Presently these services store application data in distributed data stores [LM10, DHJ+07, CDE+12, SP11, RT04], which are deployed across multiple machines in either a single or multiple data centres. On receiving a user request, the relevant application data is collected from the distributed data store, processed and returned as personalised dynamic content to users. The end-to-end latency experienced by users is measured from when the request

[6] The Cost of Latency, `http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx`, last accessed: 29/6/2013

[7] Latency Costs you Sales, `http://highscalability.com/blog/2009/7/25/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it.html`, last accessed: 29/6/2013

[8] Speed Matters (Google Research Blog), `http://googleresearch.blogspot.co.uk/2009/06/speed-matters.html`, last accessed: 29/6/2013

[9] Marissa Mayer at Web 2.0, `http://glinden.blogspot.com.es/2006/11/marissa-mayer-at-web-20.html`, last accessed: 29/6/2013

[10] Microsoft Bing, `http://www.bing.com/`, last accessed: 28/6/2013

[11] Velocity and the Bottom Line, `http://programming.oreilly.com/2009/07/velocity-making-your-site-fast.html`, last accessed: 28/6/2013

leaves the client application until the requested data from the service is returned.

Although there are various factors contributing to latency, network *propagation delay* is one of the most prominent ones. Continued increase in Internet penetration[4] and the mass adoption of online services has resulted in application data being delivered to a global user base. In 2012 the number of Facebook subscribers grew over 25% globally with key growth seen in South and Central America, Asia and Africa.[12] This added geographical distance increases the network path lengths between users and application data, and thus degrades latency performance.

Online services try to address this using distributed caching, improving the response times of global users [PF01, NFG+13]. Content is replicated onto multiple geographically dispersed servers for fast local retrieval. Content distribution networks (CDNs) such as Akamai [NSS10], CloudFlare,[13] Amazon CloudFront[14] and Limelight[15] provide this functionality as a stand-alone commercial service. Although distributed caching is essential for managing popular content, it is less beneficial for unpopular or ordinary "tail" content [And08,CKR07]. This is due to the lack of temporal locality in tail content i.e. repeated requests for the same content. Furthermore many web applications require users to have access to data for either near real-time search or dynamic computation to produce personalised content. In these cases, user requests must reach the actual application logic and *original data*, and therefore be handled by the online service's infrastructure.

An alternative method for improving data access latencies is to reduce the network path lengths between users and *all* of the content, both popular and tail. To achieve this, a service must leverage multiple DCs to provide users with application data from the "closest" DC. This has been a major motivation for popular online services to move from single DC infrastructures to multi-DC deployments. For example, Google presently comprise of over a dozen DCs.[16]

Given a multi-DC infrastructure and globally distributed users requesting for application data, two challenges arise regarding *data placement* and *data discovery*: how should such services distribute their application data across their DCs to ensure the lowest access latencies for users; and on placement of data, how do application servers find this data. While both placement and discovery strategies combined determine the latency of global online services, the number of sites in a multi-DC deployment is typically no more than a dozen. We therefore focus on data placement for this application scenario.

[12] Facebook World Stats, `http://www.internetworldstats.com/facebook.htm`, last accessed: 29/6/2013
[13] CloudFlare, `https://www.cloudflare.com/`, last accessed: 10/5/2013
[14] Amazon CloudFront, `http://aws.amazon.com/cloudfront`, last accessed: 10/5/2013
[15] Limelight, `http://www.limelight.com/`, last accessed: 10/5/2013
[16] Google Data Centers, `http://www.google.co.uk/about/datacenters/inside/locations/index.html`, last accessed: 10/5/2013

1.2 Data placement and discovery

Both of the above application scenarios have similar requirements: they (1) have globally distributed users; (2) need to provide users with low latency access to data; and (3) must return either fresh or dynamically generated data. Both applications must achieve these requirements by adopting data placement and discovery strategies that can leverage multi-site wide-area networked infrastructures appropriately. Such strategies must organise data across sites (placement) and sites into a structured network (discovery) to shorten the total network path lengths between user and data, and thus reduce latencies.

Designing effective data placement and discovery strategies is a challenging open problem. Global services and applications must understand their workloads and exploit suitable heuristics that leverage particular trends. For example, in web search engine applications such as Google, users may access data which is semantically correlated with their geographical location (i.e. has geo-spatial locality). In social network applications such as Facebook, users may access data that is correlated to what their social peers access (i.e. has social locality). Other applications may show no correlation in the semantics of content but instead exhibit high temporal locality at an individual user-level. The problem depends on the workload at hand, making it challenging to find a *general* solution that is applicable to multiple scenarios.

Furthermore our analysis in Chapter 3 shows that workloads are both complex and dynamic, with frequent shifts occurring in access patterns. Therefore any data placement and discovery configuration—i.e. a fixed placement of data and organisation of sites—that provides users with low latencies at one point in time can cease to provide low latency in the future. The problem therefore requires an *adaptable* solution that considers shifts in the workload over time.

Existing data discovery strategies range from: centralised sensor discovery platforms, such as Microsoft SensorMap [NLZ07], Xively[17] (formerly Pachube) and Open.sen.se[18] to distributed platforms, such as IrisNet [CGN05], Hourglass [SPL+04] and PIER [CSWH01]. Centralised approaches require metadata caching and provide high latencies to a global user base. Existing distributed approaches present an arbitrary random static organisation of data sources also providing users with high latencies. Many of these platforms also employ caching, improving latencies at the expense of supplying users with stale metadata.

Other strategies for organising sites into an infrastructure include DHTs [SMK01, RD01], semantic overlay networks (SON) [CGM05], and network-aware overlay networks [NZ02]. However these approaches are either unable to adapt with the workload or disregard it entirely, providing users with high data access latencies.

The situation for data placement strategies is similar. Existing global online service deployments leverage multiple DCs for increased availability, such as with master-slave repli-

[17] Xively, `https://xively.com/`, last accessed: 13/5/2013
[18] OpenSense, `http://open.sen.se/`, last accessed: 13/5/2013

cation [Aba12].[19] Drawbacks to this approach include poor write request performance and scalability, the inability to deploy small-footprint DCs, and privacy and security concerns. Other placement strategies focus on partitioning data across a single DC. Data partitioning (or sharding) is a popular practice that horizontally partitions data across multiple distributed data stores [RG00]. It improves the read and write request throughput due to the increased parallelism across instances. Among the wide range of data partitioning techniques, there is geo-spatial partitioning, which is applicable to multi-site wide-area networked infrastructure. However data partitioning techniques are static, with the data placement determined once on deployment.

Other proposals exist where the workload is periodically analysed to provide informed data placements in a WAN [ADJS10]. Although this is more accurate than static data partitioning, the drawbacks of such approaches include the periodic offline re-computation of placements. Furthermore it is unclear how the discovery of the placed data is performed in a scalable manner. Lastly, there are approaches that perform the dynamic re-partitioning of data at a single DC [PES+10, CJZM10], but those do not focus on reducing network latency.

In summary, existing data placement and discovery strategies either disregard the workload entirely or make assumptions and approximations in order to best leverage multi-site wide-area networked infrastructures. Approaches are static—unable to adapt to the workload—and therefore provide globally distributed users with high data access latencies under global and dynamic workloads.

1.3 Workload-awareness

In this thesis, we explore *dynamic* data placement and discovery strategies for multi-site wide-area networked infrastructures and propose to leverage the current request workload to perform informed online decisions. This leads to: (1) an accurate solution—providing a configuration that represents the request workload; (2) a general solution—capable of handling arbitrary trends in workloads of any application scenario; and (3) an adaptable solution—altering its configuration over time according to the current workload.

We introduce **data activity correlation** (DAC), a workload-aware approach for determining the correlations between data according to user access patterns. Correlations are defined by the similarity in user access counts between two data items. The intuition behind DAC is that data items that are frequently accessed by the same set of users are related to one another and thus correlated. For example, two webpages indexed by a web search engine that are accessed by the same users may be related because they have syntactically similar keywords, semantically related content or are listed together elsewhere on the web. The DAC approach provides four key properties:

[19] Scaling Out (Facebook Engineering), `https://www.facebook.com/note.php?note_id=23844338919&id=9445547199&index=0`, last accessed: 8/3/2013

Request locality. Highly correlated data items are those frequently accessed by the same set of users. In clustering correlated data according to the current workload—producing DAC clusters—future requests can be localised to these clusters under repetitive workloads i.e. ones with temporal locality.

Application-agnostic. Relationships between data items are determined without requiring knowledge of the underlying content. Correlations may be due to syntactic, semantic, relational, structural, geographical, temporal or social properties. For example, data items containing text in the same language/dialect may be accessed by the same set of users, but determining this relationship requires offline analysis. DAC is therefore applicable to a wide range of application scenarios.

Workload adaptability. The user access counts for determining correlations are taken from a *sliding window*, storing only the latest user requests. This allows for the correlations between data items to change according to the current workload. Coupling these correlations with an online clustering algorithm ensures that request locality within DAC clusters can be maintained over time.

Access prediction. DAC also maximises request locality by predicting the access of data. The intuition is that users are likely to access data in the future, which is highly correlated to what they have accessed in the past. This allows to pre-empt the requests of users which have never occurred before i.e. have no temporal locality.

We use the DAC approach to define two dynamic strategies, one focussing on data discovery and the other on data placement. We illustrate these strategies using the two application scenarios outlined in Section 1.1.

1.3.1 Dynamic data discovery

Data discovery strategies define the interconnection of sites and the routing of user requests across this multi-site networked infrastructure. Their goal is to reduce the latency experienced by users when accessing data by reducing the number of hops performed. The DAC approach can be applied as a strategy by organising sites into a structured network that is *workload-aware*. Sites are represented by the data that they store and arranged into a hierarchical clustered network, with correlated sites co-located in the same cluster. Users are assigned to the clusters that have satisfied the most requests in the past. The intuition is that subsequent user requests can be handled by assigned clusters and thus requiring fewer hops.

The multi-site networked infrastructure *adapts* according to the request workload with an online distributed algorithm. While the correlations between sites evolve with the workload, it is the algorithm that ensures that these changes affect the network structure. This algorithm is divided into two disjoint components: the *split* component, which partitions clusters into two; and the *merge* component, which combines two clusters to form one. Both components maximise the correlations between the sites arranged into clusters through their

respective processes. The combined effort of the DAC approach and the *split-and-merge* algorithm ensures that the multi-site networked infrastructure is both workload-aware and self-organising.

We explore this data discovery strategy with Global Sensor Space (GSS), a global sensor discovery platform that interconnects sensor data sources into a single unified infrastructure. The discovery strategy is applied with data sources representing sites that store the sensor metadata. GSS provides users with low latency access to sensor metadata that is fresh and can support arbitrarily complex queries.

1.3.2 Dynamic data placement

Data placement strategies define the placement of data across multi-site networked infrastructures. Their goal is to reduce the latency experienced by users when accessing data by shortening the network path lengths between them. The DAC approach can be applied as a strategy by performing the workload-aware partitioning of data into clusters, with correlated data items co-located in the same cluster. Clusters are placed in sites that are geographically closest to their requesting users. The intuition is that subsequent user requests can be handled by these sites, thus reducing latency during data access.

The placement of data across a multi-site networked infrastructure *adapts* according to the request workload with two online distributed algorithms. While the correlations between data items evolve with the workload, it is these algorithms that reflect these changes in the placement of data. The *workload-aware data partitioning* algorithm dynamically re-organises data across clusters to ensure that the data items within clusters remain correlated. The *geo-aware partition placement* algorithm dynamically places clusters across sites according to the geographical locations of their requesting users. The combined effort of the DAC approach and the two algorithms ensures that the placement of data throughout a multi-site networked infrastructure is both workload-aware and dynamic.

We explore this data placement strategy with Skyler, a geo-dynamic partitioning middleware for global online services, which optimises the placement of application data across multiple DCs. The placement strategy is applied to this application scenario with DCs representing sites and partitioning representing clusters. Skyler provides globally distributed users with low latency access to the original application data, i.e. not cached or replicated.

1.3.3 Global workload generator

To evaluate both GSS and Skyler under a variety of request workloads, we present a configurable wide-area network workload generator. The general lack of global-scale user request datasets in the public domain has made the evaluation of multi-site wide-area networked infrastructures a hard problem. Furthermore, existing workload generators [Coo10, LLSG07, KXP12] focus on modelling popularity, read-write ratios and inter-arrival rates of requests, ignoring the geographical distribution of users and their relationship to requested data.

The workload generator simulates the requests of globally distributed users with configurable geo-spatial locality properties. These properties can be defined to either remain constant, producing a static workload, or change over time, providing a dynamic workload. The static variant expresses the geographical radius of interest in data items and the displacement of this radius. The dynamic variant models *shifts* in the geo-spatial locality properties by varying the placement of the radius over time. Shifts are modelled according to changes in the popularity of data [RFF+10].

1.4 Research contributions

In this section we summarise the contributions made in this thesis, presenting them in the same order in which they appear:

- **Data activity correlation.** The workload-aware approach determines the correlations between data items based on the access patterns of users. The approach presents a number of key properties, which we use to provide workload-awareness to two data placement and discovery strategies.

Applying the DAC approach as a strategy for *data discovery* provides:

- **Workload-aware network.** The DAC approach is used to organise sites and the application-level links between them to provide a structured network based on the access patterns of the past workload.
- **Split-and-merge algorithm.** The workload-aware network *adapts* its structure according to the request workload with a distributed online algorithm, which actively optimises for the reduction in hops—and thus user response time—during the discovery of data.

Applying the DAC approach as a strategy for *data placement* provides:

- **Workload-aware data partitioning.** The DAC approach is used to dynamically organise data across partitions according to the access patterns of the past workload.
- **Geo-aware partition placement.** The geo-aware placement of partitions across a multi-site wide-area networked infrastructure is performed dynamically and according to the geographic locations of their requesting users.

The evaluation of our dynamic data placement strategy provides the final contribution:

- **Global workload generator.** We introduce a configurable static and dynamic workload generator for the evaluation of multi-site wide-area networked infrastructures. The generator simulates the requests of globally distributed users with a wide range of geo-spatial locality properties.

1.5 Dissertation outline

The remainder of this dissertation is organised as follows:

Chapter 2 provides the relevant background material. The chapter begins with a detailed problem analysis followed by existing data placement and discovery solutions for the two application scenarios that we have investigated. We then outline the relevant material that is in the distributed data management domain, focussing on data partitioning, replication and transactions. We end this chapter with a discussion of the state-of-the-art for data placement and discovery strategies.

Chapter 3 introduces *data activity correlation* (DAC), our workload-aware approach for identifying correlated data. We begin the chapter by performing an analysis of current workloads for global online services and applications, investigating the various factors involved in user-data interactions. This provides the motivation and intuition to introduce our DAC approach. Following this, we perform an experimental exploration of DAC on both synthetic and real-world datasets. We end this chapter with a discussion of DAC and its relevance to data placement and discovery.

Chapter 4 presents a dynamic data discovery strategy applied to one application scenario. This chapter introduces Global Sensor Space (GSS), a global sensor discovery platform with a workload-aware and self-organising network of data sources. The chapter begins by applying the DAC approach to the data discovery problem. We then provide the design for GSS, followed by our online distributed clustering algorithm. We end the chapter with a simulated evaluation of GSS, comparing it with a static approach under both geo-spatial and semantic locality workloads.

Chapter 5 presents a dynamic data placement strategy applied to one application scenario. This chapter introduces Skyler, a geo-dynamic partitioning middleware designed to leverage multiple DCs for the reduction of data access latencies. The chapter begins with the system design, followed by three sections outlining the components of the system: a workload-aware data partitioning algorithm that is based on the DAC approach; a geo-aware partition placement algorithm that migrates data partitions to improve their access latencies; and various data discovery strategies for forwarding user requests toward the dynamically placed data. We end this chapter with an evaluation of Skyler, comparing it with existing solutions under both static and dynamic workloads.

Chapter 6 draws conclusions with regard to what has been presented in this thesis, in addition to outlining future research directions.

Chapter 2

Background

In this chapter we provide relevant background knowledge for the problem and solutions outlined in the rest of this thesis. As this work spans a variety of research topics, we focus on work in both industry and academia related to the management of data in distributed systems in terms of data placement and discovery in multi-site wide-area networked infrastructures.

The content of this chapter is divided into six sections. Section 2.1 provides a detailed analysis of the problem of reducing latencies in global online services and applications from an abstract perspective. We then outline its concrete requirements by describing two application scenarios (Section 2.2). In Section 2.3 we give a brief overview of the topics related to distributed data management such as partitioning and replication. Following that we describe the state-of-the-art in data placement (Section 2.4) and data discovery (Section 2.5). We finish with a summary of the chapter in Section 2.6.

2.1 Problem analysis

Global online services and applications have high-level actions for users to execute. These actions provide a layer of abstraction that bridges the gap between the functionality of an application and the low-level access and processing of data. For example, a micro-blogging service may provide users with an action to follow another user, resulting in a notification on any future blog posts. On execution, actions are encapsulated on the *front-end application* in user requests, which are sent across the network to the *back-end application* running on the service infrastructure. User requests are processed according to application logic, performing potentially multiple data access operations—i.e. *data requests*—on a given data store. User requests that result in multiple data access operations are *multi-get* requests.

The accessing and manipulation of stored data is an essential component for global online services and applications to provide users with functionality. Large-scale networked applications must organise stored data in a multi-site distributed system, with the specifics dependent on each application scenario. For example, a global sensor discovery platform stores sensor metadata across thousands of Internet-connected data sources. Conversely, a global online

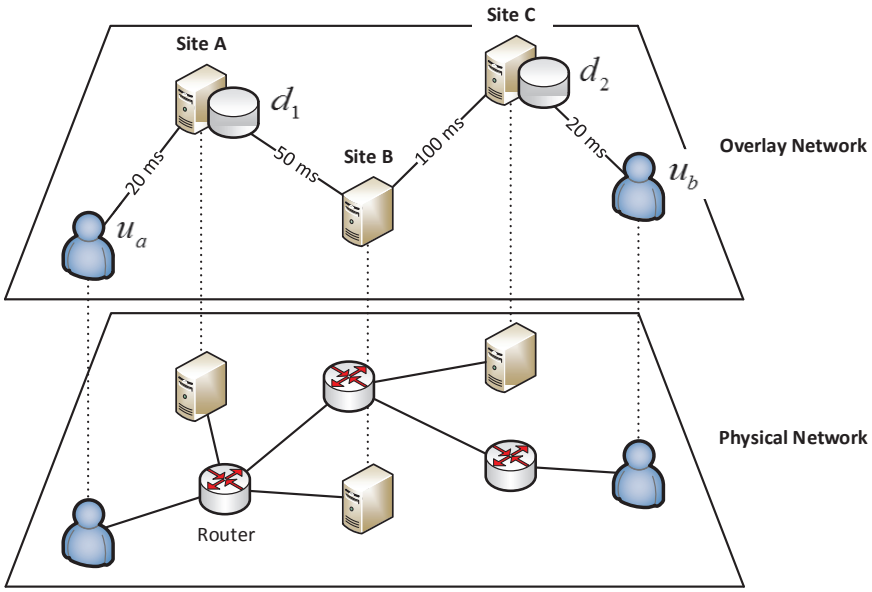


Figure 2.1: Example configuration of sites, data items, users and application-level links, and the overlay network mapping onto the physical network. Links between entities show one-way network latencies in milliseconds

service such as Facebook or Google stores application data across a dozen data centres, each consisting of thousands of data store instances. Although application scenarios may differ in both the number of sites and their data management capabilities, applications share a common goal: to provide globally distributed users with low response times when accessing data, which is distributed across a multi-site wide-area networked infrastructure.

To illustrate this goal, we provide an abstract model of the problem. In this model, each user requests a single data item, which is assigned to a distinct site. The term "data" item is loosely defined to encompass multiple application scenarios, including sensor metadata, database table rows and data store key-value pairs. We give more detail in Sections 2.2.1 and 2.2.2 where the problem is stated concretely for two application scenarios.

Figure 2.1 shows an example *configuration* of a multi-site wide-area networked infrastructure comprising of sites A-C, with sites A and C storing single data items, d_1 and d_2, respectively. A configuration is an arrangement of sites, users, application-level links and the placement of data items across sites. The application-level links between sites form a network, known as an *overlay network* [LCP05], which maps onto the physical network. Sites in an overlay network are also known as *nodes*. The traversal of a link on an overlay network—i.e. an *overlay hop*—typically involves the traversal of multiple routers and physical network links—i.e. *network hops*. We use overlay hop and hop interchangeably throughout this thesis.

The figure shows users u_a and u_b with a one-way network latency of 20 ms to their "closest" sites, sites A and C, respectively. Proximity can be defined in terms of latency or more commonly approximated by the number of network hops, geographic distance [KPL+09] or the overlap between IP address prefixes of two hosts [FVFB05].

We consider two workloads, W_1 and W_2, when describing the problem at hand:

Workload W_1 involves users u_a and u_b retrieving data items d_1 and d_2, respectively. Users send their requests to the closest sites, which in turn retrieve the required data and return it as a response. The entire interaction has a round-trip network latency of 40 ms for each user. Under such a workload, the current configuration is optimal because all data is provided to users by the closest site. This assumes that all sites and links are equally loaded.

Workload W_2 involves users u_a and u_b retrieving data items d_2 and d_1, respectively. In this scenario, users issue a request to their closest site, which must then be forwarded through site B to obtain the required data item. The additional application-level routing results in a network latency of 340 ms for both users. Under this workload, the current configuration is sub-optimal because data is provided to users by a remote site, with two additional network paths resulting in a higher latency.

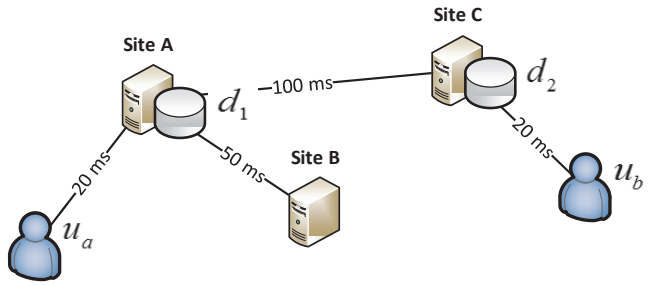
In general, user response time is measured from the moment a user submits a request on a front-end application to when a response is returned to the user. It can be broadly divided into three components:

1. **Processing time** is the amount of time taken to process the request/response at both the front-end application and the service infrastructure. This may involve the execution of application logic, the fetching of data from storage or the presentation of data. The time taken to process a request is therefore affected by factors such as hardware capabilities, computational load and software architecture.

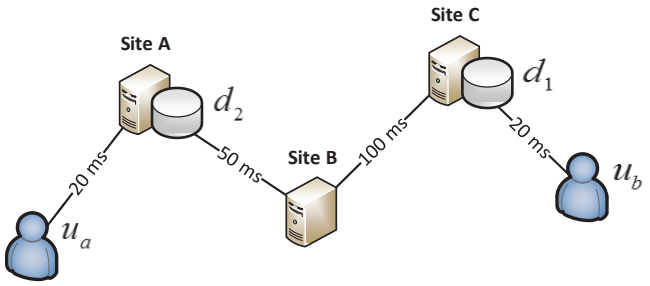
2. **Network latency** is the delay incurred in communicating a request/response over a *network path*. It is influenced by factors such as the propagation delay, data protocols, packet routing and switching, and queuing and buffering [LGW03]. We assume that communication is performed over the Internet through a reliable connection-oriented protocol such as TCP/IP.

3. **Application-level routing** is the delay due to a request being re-routed to another site in the infrastructure at the application layer. This routing is performed on an overlay network and therefore only applicable to multi-site infrastructures. Each application-level routing action involves additional processing time at a site.

An effective method to improve network latency and therefore user response time is to shorten the total network path length between the user and a data item. Shorter network paths can improve the Quality-of-Service (QoS) [BP08, OA04] by offering a number of benefits: (1) *lower latencies*—shorter network paths reduce the propagation delay; (2) *higher reliability*—routing data through fewer network elements and links improves reliability as the chances of failure are reduced; (3) *improved throughput*—with fewer network elements and links, paths are less likely to encounter congestion, thus offering improved throughput; and (4) *decreased network saturation*—by keeping communication local, fewer parts of the network



(a) Improved data discovery strategy



(b) Improved data placement strategy

Figure 2.2: Two improved data discovery and placement strategies for the scenario from Figure 2.1

are used, thus decreasing overall network saturation and leaving more network capacity for other traffic.

Figure 2.1 illustrates how the configuration of sites, users, application-level links and data items is critical to the network path length of requests under a given workload. We can therefore reduce network latency by generating better configurations. Intuitively this can be achieved by adopting appropriate **data placement** and **data discovery** strategies. We describe these strategies under workload W_2: users u_a and u_b retrieving data items d_2 and d_1, respectively.

Data discovery. Figure 2.2(a) illustrates an alternative configuration to the one shown in Figure 2.1, which results in a network latency of 240 ms for both users. The configuration is generated with an improved data discovery strategy, which involves maintaining an application-level link between site A and site C. The general strategy adopted is to reduce the number of hops performed by constructing the appropriate overlay network for a given workload.

Data placement. Figure 2.2(b) illustrates an improved data placement strategy. In this strategy, the data items accessed by users are located at the user's closest sites, with site A storing data item d_2 and site C storing data item d_1. This provides the lowest network latency with both users accessing their respective data items in 40 ms. The general strategy adopted is to reduce the number of hops performed on the overlay network by placing the appropriate data items across sites for a given workload.

There is a clear relationship between the placement and discovery of data on multi-site

wide-area networked infrastructures. When adopting a sophisticated placement strategy, the strategy for discovering data can be rudimentary such as always forwarding requests to the next closest site; similarly when performing a sophisticated discovery strategy, the placement of data remains important but less critical. For example, the improved placement strategy shown in Figure 2.2(b) does not require the forwarding of requests beyond the user's closest sites.

Identifying an optimal configuration in the illustrated problem is simple due to the small number of sites, data items and users. However, designing data placement and discovery strategies to generate good configurations in a scalable and distributed manner is a challenging open problem. This is because of the large solution space with any reasonable number of entities. The complexity of the problem is further increased with a *dynamic* workload: even after an ideal configuration is found, the workload may change over time, with the configuration becoming outdated and providing progressively worse performance.

2.2 Application scenario

This section describes the problem statement of reducing network latencies in multi-site wide-area networked infrastructures for two application scenarios, each with different requirements and data management capabilities. The first scenario involves thousands of globally distributed sensor data sources, each storing their own metadata. The second scenario involves a dozen globally distributed data centres storing the application data for global online services such as Facebook and Google. It is these differences that forces global services and applications to leverage the relationship between data placement and discovery.

2.2.1 Global sensor discovery

The "Internet of Things" (IoT) [Ash09] is a paradigm that calls for the world's sensors and actuators, which are seamlessly embedded throughout our environment, to communicate amongst themselves, users and autonomous applications. The motivation for this is to bring major advancements to applications and domains such as smart environments (cites, homes, plants), transportation and logistics, environmental monitoring, healthcare, and security and surveillance [MSPC12, AIM10].

To assist with this, we provide our own interpretation of the IoT vision: it needs a distributed planet-wide sensor collaboration and processing system that interconnects the world's sensor data sources. We envision such a system to be capable of gathering, filtering, processing, analysing and disseminating streams of sensor data from global data sources in real-time. This must be performed according to the submitted requests of globally distributed users and autonomous applications.

A critical requirement for such a system is the ability to *discover* relevant sources of sensor data. We therefore focus on the problem of developing a **global sensor discovery platform**

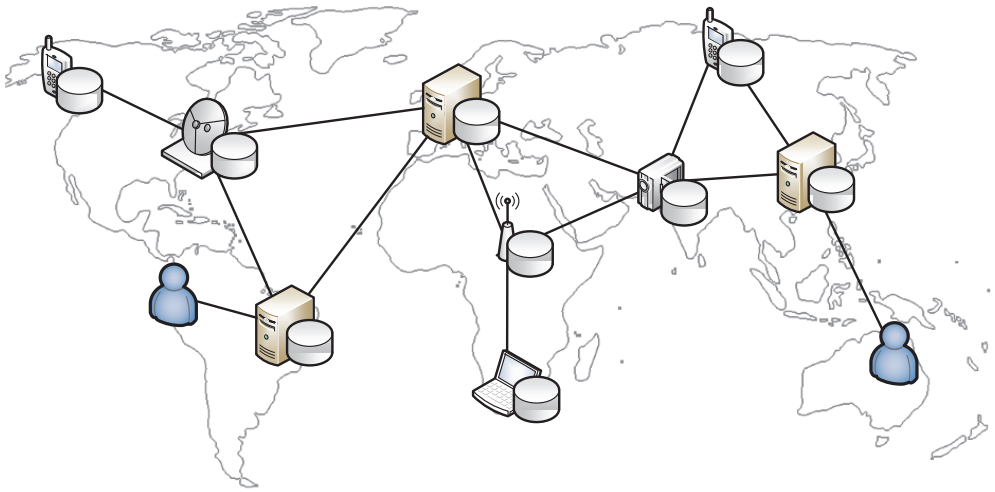


Figure 2.3: Example of a global sensor discovery platform with interconnected users and sensor data sources storing metadata

that interconnects the planet's data sources into a single unified infrastructure. The aim of the platform is to execute requests by selecting sensor sources from the large number of globally distributed Internet-connected data sources available, while achieving low response times.

A global sensor discovery platform aligns with the previously defined abstract model in Section 2.1. Each data source is a site storing individual data items—i.e. *metadata* of sensor data sources. The organisation of sites and their interconnection over the Internet is an overlay network. The goal is to provide users and applications access to this metadata with low latency.

Figure 2.3 illustrates an example global sensor discovery platform consisting of various globally distributed sensors and users. Each sensor data source on the platform stores its own metadata consisting of: static attributes such as sensor type, model and measured units; and dynamic attributes such as the last measurement, average measurement over a timespan and location (if mobile). Dynamic attributes are updated continuously with the latest measurements taken such as a temperature reading.

Users issue multi-attribute requests (or queries), which are executed against the data source metadata. For example, a user may request for the temperature sensors in Europe which are currently above 20 °C. For *accurate* results, the data source metadata must be *fresh*: an outdated measurement of 21 °C when currently 19 °C, leads to a false positive result; similarly an outdated measurement of 19 °C when currently 21 °C, leads to a false negative result. Accuracy at one point in time is therefore dependant upon the *delay* between taking a measurement and updating the data source metadata. This delay is accentuated with a longer network path length between the sensor and metadata.

This presents a trade-off between accuracy of results and the user response time provided by a platform: (1) for the freshest results, requests must reach the metadata directly stored

at the data source itself; and (2) for network efficiency, data source metadata can be cached at more sites in the infrastructure, thus reducing the number of hops performed over the overlay network. While caching metadata increases the network and computational efficiency of a platform, it requires a platform to handle the consistency of replicas, the invalidation of cached copies and any privacy or legal constraints. Caching is a critical feature for the scalability of distributed systems with skewed popularity distributions, efficiently balancing the read request load of highly popular content.

A key component to such a platform is the query model adopted and the expressiveness provided to users. A limiting platform relies on the user to submit *exact-match* queries, or also known as a "simple query" [Knu73], "point search" [FB74] or a "get" operation, which point to a single source or data item. Under such a model, users must know the data items that are stored before they can be successfully queried. Other broader query models provide users with a more powerful and expressive query language to select sources or data items. As we illustrate throughout this section and Section 2.5, there is a dependency between the data discovery strategy adopted and the query model provided by a platform.

In the following sections, we discuss relevant existing research work on large-scale sensor discovery and processing platforms, in addition to the techniques used to perform data source discovery. We are specifically interested in how discovery or search of data sources is achieved, the properties and assumptions made and the response times users or applications can expect with such systems.

Centralised sensor data discovery

Sensor discovery is an active topic in both research and industry, with many services and infrastructures having been proposed and developed. Industry has mainly focussed on *centralised* approaches, with services such as Xively[1] (formerly Pachube), Open.sen.se[2] and Microsoft SensorMap [NLZ07] being the most commonly known. They essentially adopt a "search engine" approach to discovering sensor information. This has disadvantages, which we outline below:

Metadata freshness. A challenge of a centralised service is to keep the metadata of millions of sensors up-to-date with the latest sensor availability and readings. If a system only presents static attributes to users, it restricts the requests that can be issued reducing its functionality; conversely if the system provides both static and dynamic attributes, it has to contact all sensors periodically (pull-model) or have the sensors send the latest changes periodically (push-model). This delay between the sensor measurement and the updating of its metadata affects the accuracy of results. The separation between the sensor and its metadata also places the access control and security restrictions of sensor data in the hands of the centralised service.

[1] Xively, `https://xively.com/`, last accessed: 13/5/2013
[2] OpenSense, `http://open.sen.se/`, last accessed: 13/5/2013

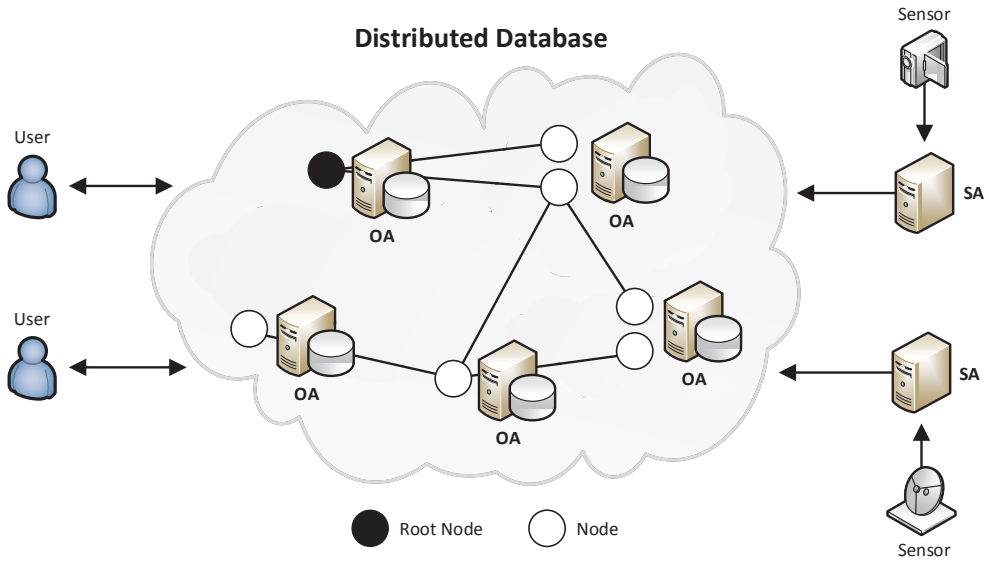


Figure 2.4: IrisNet architecture, which consists of sensors, Sensing Agents (SA), Organising Agents (OA) and users

High network latencies. A centralised discovery approach limits the system's ability to exploit the geographical distribution of sites to provide globally distributed users low latency access to sensor metadata. All requests must instead be directed to a central site, irrespective of the geographical location of users.

Decentralised sensor data discovery

The research community have instead considered the development of global sensor discovery systems deployed over a distributed set of sites [CGN05, DNGS03, SPL+04, HHL03]. The following sections review these systems and their shortcomings in providing fresh data to users with low response times.

Internet-scale Resource-Intensive Sensor Network Services (IrisNet) [CGN05,DNGS03] is one of the earliest sensor discovery systems with the aim of simplifying the development of sensor applications that require the discovery of sensors. Its infrastructure is based on a two-tier architecture consisting of:

1. *Sensing Agents* (SA), which collect and pre-process streams of sensor data, extracting the desired information from nearby sensors. Sensor agents are powered Internet-connected PCs.

2. *Organising Agents* (OA) or sites, which are interconnected to provide a distributed hierarchical database. OAs manage partitions of the database and store processed data from SAs. OAs are tasked with storage and querying.

The system makes use of a hierarchical XML data model for its distributed database and uses DNS-style [MD88] naming to route queries toward partitions of the database. It uses a

technique called *self-starting distributed queries* which directs queries to the lowest common ancestor (LCA) of the hierarchy by using DNS-style site names, which are extracted from the query itself. This ensures that the OA hosting the root node is not overloaded with queries. The nodes of the hierarchical database are distributed across the available OAs.

Figure 2.4 illustrates the IrisNet architecture, which consists of sensors, sensing agents, organising agents and users. SAs receive streams of data from sensors and trigger updates to the distributed database. The OAs store partitions of the distributed database consisting of nodes in the hierarchical XML data model. The figure shows the partitioning of the database with the XML data model nodes distributed across OAs. An example *schema* for the data model is to arrange data according to geographical location: the root node defined as the "World", its children as regions such as the "US", "Europe" etc. , and so on. OAs also provide users with access to the data that is stored.

With DNS-style naming, IrisNet performs DNS lookups to obtain the IP addresses of the relevant sites during the execution of a query. This can potentially lead to multiple overlay hops. DNS results are heavily cached to optimise subsequent DNS lookups to this same site. Once a query reaches a site, the local database is searched: upon success, the entry is returned; or upon failure, subsequent queries are performed. Finding the IP addresses of subsequent sites is achieved by constructing the relevant DNS name from the query and the site's host name.

While DNS lookups are used to obtain the IP addresses of OA or sites, a *query-evaluate-gather* (QEG) technique is used to detect: (1) what data is relevant to a given query at a particular site; and (2) how to retrieve the missing parts of the data. This is an essential part of the system because sites store partitions of the database. The QEG mechanism determines when, for example, subsequent queries must be issued because the local database has insufficient information to answer the query.

The system makes heavy use of caching, with each OA capable of holding various granularities of data belonging to other OAs. OAs adopt a *query-driven* caching mechanism, which places copies of data close to users who have previously queried a sensor. OAs therefore aggressively cache query results. IrisNet adopts a policy of *never* removing cached data and only updating it when new queries are issued. DNS lookups are also cached to ensure that the logical to physical mapping can be done efficiently.

Such a design has a number of shortcomings:

Arbitrary logical-to-physical mapping. Each site stores arbitrary partitions of the hierarchical data model, which consists of multiple nodes. The mapping from logical nodes in the hierarchical database, to physical sites has therefore no correlation. IrisNet does not define how database partitions are distributed across OAs. Therefore traversing the logical structure—i.e. overlay network—can result in excessive application-level routing time due to a lack of network path locality. The authors provide an example where it is possible for a parent node and grandchild node to be placed on one SA whilst the child nodes are placed on another. In general, the overlay network constructed by IrisNet is independent from the

underlying network.

Hierarchical structuring of data. Not all sensor data can be structured hierarchically. While, for example, geographical locations provide a way to partition sensors and their data, sensors such as RSS or social network feeds have no inherent physical location. These data sources may be better suited toward, for example, *topic-based* or *semantic-based* categorising approaches [CGM05] which we discuss further in Section 2.5.5. Sensors unable to fit the hierarchical structuring imposed by IrisNet are likely to have high access latency due to the excessive routing required during discovery.

Freshness of logical-to-physical mapping and data. When mapping queries to sites, resolved host names are cached locally. While this can answer queries with low latency, it has the risk of supplying stale data. Using stale data to resolve a host name adds an additional delay, affecting the total network latency for responding to a query. In addition, IrisNet's query-driven caching mechanism, replicates database partition onto various OAs. Query results can therefore also be inaccurate as sensor metadata may be stale.

Inflexible query model. The hierarchical XML-based data model provides a single-attribute index, which is then distributed across OAs. Queries must follow the hierarchical naming convention of sensors, always including the indexed attribute to perform fast and efficient retrieval, known as an *exact-match* query. The authors do not discuss how complex queries are handled i.e. when a non-indexed attribute is provided within a query. This most likely results in a full tree search from root to all leaves resulting in long network paths and thus high network latencies.

Hourglass [SPL+04] provides an infrastructure for interconnecting sensors, services and applications over the Internet. It defines a *data collection network* (DCN)—an overlay network of collaborating hosts, which supports the flow of data from multiple globally distributed data sources to applications. The Hourglass system divides components into *producers*, *consumers* and *services* and focusses on the quality-of-service (QoS) of data streams in the light of disconnections. It achieves this by establishing and maintaining *circuits* in the overlay network.

Services perform filtering, aggregation, compression and buffering of data along *circuits*. Circuits are constructed by a *registry*, which maintains resource availability information, and a *circuit manager*, which assigns links and services to actual nodes that are within the overlay network. The circuit manager creates the circuits thus determining the latency involved in data discovery. However no details are given on the policy governing circuit creation.

Hourglass is *topic-based* and partitions the space of available services in the DCN into distinct mutually-agreed-upon topics. It is Hourglass' users that define the topics that are available in the system and therefore not as restrictive as, for example, a hierarchical-based organisation such as with IrisNet. Applications can generate *unrealised circuit* with an Hourglass Circuit Descriptor Language (HCDL) consisting of topic and predicate statements i.e. queries for identifying the relevant services. It is the task of the circuit manager to realise these circuits

by using the registry as a lookup service, matching the topic and predicate statements to services.

For the purposes of this thesis, we are interested in the latency properties of Hourglass when performing sensor lookups. The disadvantages to Hourglass' infrastructure with regards to the discovery of sensors are outlined below:

Unknown logical-to-physical mapping. A major component of the Hourglass system is the registry, which acts as a distributed lookup service. Circuit managers forward the statements of unrealised circuit descriptors to the registry, which responds with the corresponding service providers. How this registry is implemented is left open by the authors, with potential solutions being cited as DNS [MD88] and DHT-based systems Pastry [RD01] and Chord [SMK01] (see Section 2.5.4). Such solutions, however, would provide stale metadata or an arbitrary logical-to-physical mapping, leading to high response times for lookups.

Inflexible query model. Hourglass operates an exact-match query model similar to that of IrisNet. Queries can be viewed as the submission of an unrealised circuit, which must be realised by a circuit manager. It must use the given topic and predicate statements in the circuit descriptor to map service providers to applications. Service providers with the exact same statements are matched with the descriptor to provide this mapping. Generating unrealised circuits requires knowledge of the available topics and predicated to ensure circuits can be mapped onto the correct services. This is a limitation of Hourglass and the functionality it provides to applications.

Peer-to-Peer Information Exchange and Retrieval (PIER) [HHL03] is a large-scale query engine for applications that perform *continuous queries* over streams. The system has a three-tier architecture: (1) *Applications* interact with the (2) PIER *Query Processor* (QP), which in turn uses a (3) *Distributed Hash-Table* (DHT) as its underlying overlay network structure (see Section 2.5.4). DHTs are a widely used mechanism for performing exact-match queries with a low network hop count, but decouple data from where it is generated. In PIER, the DHT is used as an indexing abstraction, decoupling sensor data sources from their indexed metadata and storing an address to data sources. We review DHTs further, outlining their properties and limitations, in Section 2.5.4.

PIER's focus is on continuous query processing and therefore lacks the ability to discover fresh data sources with low response time. Its DHT-based mechanism for data discovery has the disadvantages outlined below:

Arbitrary logical-to-physical mapping. DHTs provide a random mapping from the logical structure (i.e. a key space) to the physical network. Although routing in a DHT provides a low (and theoretically bounded) hop count in the overlay network, the underlying network path length may be large. In PIER, the DHT is used to store values consisting of the IP address of services.

Freshness of stored keys. PIER uses a DHT for the lookup of resources based on a key, which is the hash of a given *namespace* and *resourceID* for an object. ResourceIDs

provide the semantic meaning of an object and the namespace acts as a logical grouping. A disadvantage of this approach is that, once hashed keys are inserted in the DHT, they remain static, with resourceIDs unchanged according to the latest data produced.

Inflexible query model. DHT-based systems have an exact-match query model. Key-value pairs can be inserted and looked up. In PIER, queries must therefore provide the correct resourceID and namespace to generate a key that retrieves the IP address of the required service. This limits the search and retrieval functionality of a system for sensor discovery because it can only represent a single static attribute in queries.

2.2.2 Global online services

Global online services such as Facebook, Twitter and Google all share the common goal of providing globally distributed users with low latency access to stored application data. User response time is a major concern for services, with milliseconds in added delay directly translating into loss of revenue (see Section 1.1.2).

Services have therefore moved from single DC infrastructures toward multi-DC deployments, leveraging the geographical location of individual DCs to reduce network path lengths. Facebook opened a new DC in Virginia in 2008 with the primary reason of gaining 70 ms in latency for their users on the US east coast and in Europe.[3] However, with this shift comes a new set of open challenges, the most prominent being how to place application data across these DCs to ensure that users experience lower response times when interacting with the service.

At a first glance, the problem seems trivial: application data can be assigned to DCs that are closest to the original content creator. For example a user's social networking profile should be placed in a DC that is geographically close to the user to ensure that they experience low latency access. The problem, however, is more complex due to the properties and variations of each service's user request workload. User-generated content (UGC) may be accessed more often by users other than its content creator. Furthermore, some application data may not have an identifiable creator such as web pages that are crawled and indexed by online search engines.

Consider the case of an online social network with a social graph, imposing relationships between users and the data that they access and produce. By placing a user's data in one DC and the data of a social connection in another, a request for the latest activity in the graph by either user requires the retrieval of application data from multiple DCs. This adds additional inter-DC network latency resulting in higher user response times and degraded user experience. A good data placement strategy should therefore be conscious of the workload, which in turn is affected by a multitude of factors. We discuss these factors in further detail in Section 3.1.

In the following sections, we describe how existing online services operate, the functionality

[3]Scaling Out (Facebook Engineering), `https://www.facebook.com/note.php?note_id=23844338919&id=9445547199&index=0`, last accessed: 8/3/2013

Name	Type	Description
id	int64	unique identifier
type	int32	object type
version	int64	object version
update_time	int32	last modification
data	text	data

Name	Type	Description
id1, id2	int64	edge endpoint IDs
atype	int64	association type
visibility	int8	edge visibility mode
timestamp	int32	client sort key
data	text	data

Table 2.1: Facebook database schema for *objects* (left) and *associations* (right) [AP13]

that they offer to users and the application data that they store. We discuss the data placement strategies that they adopt for both individual DCs and multi-DC deployments. We also explore how these various designs affect user response times. Unfortunately, in many cases, the details of these services are either not widely publicised or kept as trade secrets. We therefore gather as much information and knowledge as we can from various scientific publications, industry presentations and technical blogs.

Online services

Facebook is a popular online social networking service with over 1 billion monthly active users [Fac12]. To participate in the service, users create a *profile* with their personal information. They can then connect with other users throughout the service by establishing *friendships*. Interaction between friends is achieved by posting content (both text and multimedia) onto their own or friends' *walls*. Each wall belongs to a user and is a shared space for user interaction. All posts can be modified by other users with comment or 'like' tags. Finally, users are shown a personalised timeline called a *news feed*, which displays the latest posts made by their personal social network of friends.

At the underlying data management level, Facebook's social graph is made up of multiple *objects*, representing graph nodes; and *associations*, which are directed edges between objects [AP13]. Objects can be of a variety of entity types, such as user profiles, status updates, photo albums, or photo and video metadata. Associations represent relationships between the objects the graph, such as a user's friendship with another user, a user posting a photo or video, or a user liking a status update. The database schema used to store these two entities is shown in Table 2.1. The unique key is the *id* field for the object entity and the (*id1*, *atype*, *id2*) fields for the association entity.

Facebook's social graph is stored using MySQL (InnoDB) [BZ05], a relational database management system (RDBMS). However, the company claims not to use it for its relational properties but rather for its stability and predictable behaviour in persisting data.[4] We discuss RDBMSs and NoSQL data stores and their differences further in Section 2.3. Due to

[4] High Performance at Massive Scale (Facebook), `http://cns.ucsd.edu/lecturearchive09.shtml#Roth`, last accessed: 10/11/2012

the size of the social graph, data is partitioned into *partitions* or *shards* and stored in multiple MySQL instances across machines.[5] Data partitioning is a well-known technique used for improving scalability when vertical scaling techniques are no longer possible or effective.[6] We discuss partitioning in more detail in Section 2.3.1 and the various schemes in Section 2.4.1.

Initially, when Facebook membership was restricted to only a dozen universities, data was partitioned across machines in one site according to the a user's affiliation to a university.[4] The majority of user's social connections were from their own university. As a result, a user requesting for the latest posts amongst their connections would translate to a data request issued to a small number of partitions on machines. This provided high *partition isolation*, avoiding multi-partition requests and thus reducing network utilisation [SC11]. Partition isolation also improves response times by avoiding the wait for the slowest machine to return results. We discuss isolation further in Section 2.3.1.

Although this was a good partitioning scheme at first, Facebook's underlying workload changed. Members graduated and moved to other universities for postgraduate studies, making new friends and social connections. Facebook began opening up its service to members from more universities and eventually the general public. The end result of these changes is an increasingly *tangled* social graph. Facebook's strategy to partition users according to their network soon became a problem. Requests for a user's latest news feed now resulted in accesses to many database servers. Partition isolation had vanished due to the dynamic nature of the workload.

Currently, Facebook partitions data using a custom *consistent hashing* scheme [KLL97], which groups related objects together and places unrelated objects randomly across partitions. Although this is a more predictable strategy than the previous one, Facebook are unable to obtain the high isolation they once had.

Twitter is a micro-blogging service with over 200 million monthly active users.[7] Unlike other social networking services such as Facebook, relationships between users in Twitter are directed: users can follow any other user without the need for them to follow back. A user's followers receive *tweets*, which are posts consisting of a maximum length of 140 characters. Tweets can contain *hashtags*, which are words prefixed with the '#' character, providing context to tweets. Hashtags allow for users to discover other tweets made in the same context such as a location, object or event. Twitter also provides users with mechanisms to interact such as *retweeting* an existing tweet, i.e. propagating it to followers, or addressing a particular user in a tweet.

Each Twitter user has a personal timeline according to which users they follow. Tweets from a timeline are ordered according to their time of posting. Through the timeline, users are

[5]Facebook MySQL Conference 2011, `http://www.slideshare.net/ryanthiessen/mysql-conference-2011-the-secret-sauce-of-sharding-ryan-thiessen`, last accessed: 12/2/2013

[6]Early Amazon: Splitting the website, `http://glinden.blogspot.co.uk/2006/02/early-amazon-splitting-website.html`, last accessed: 4/7/2012

[7]Twitter Official Status, `https://twitter.com/twitter/status/281051652235087872`, last accessed: 20/4/2013

able to *reply* to tweets directly, *retweet* an existing tweet or *favourite* a tweet, i.e. expressing their interest in a tweet.

Similar to Facebook, Twitter uses MySQL (InnoDB) to store both the relationships in the social graph and the tweets. Data is also partitioned across machines within a DC through a middleware system, Gizzard,[8] located between the application and the MySQL instances. Twitter's workload is temporally skewed, with the vast majority of requests issued for the latest tweets (see Section 3.1.4). Their initial strategy was to partition data according to when the tweet was produced (*timestamp*) in order to provide high partition isolation.[9] A small number of partitions stored the latest tweets, which would therefore handle the majority of the current workload. To support the large request traffic on these partitions, Twitter provisioned them with many replicas, thus balancing the load.

With Twitter's fast paced growth, it soon became apparent that this was not a scalable approach. Temporal partitioning quickly led to the degradation of write request throughput due to the latest master partition being stored on a single machine. Twitter has since evolved its infrastructure and opted for a more load-balanced approach, combining Gizzard with consistent hashing to partition data but again sacrificing isolation.

These services have tried a variety of strategies to place or partition data across multiple machines within a DC. In the following paragraphs we discuss how online services operate and manage their data across multiple data centres and other geographically distributed infrastructures. The goal in either case is to provide users with low latency access to their application data.

Both Facebook[3] and Twitter report to be using a full master-slave (M-S) replication model, with the total number of replicas matching the number of sites. In M-S replication a single master copy of the data is held at one data centre, with slaves on all other DCs. It can leverage the geographical location of DCs to provide nearby users with low *read* request latency. *Write* requests, however, must be handled by the master DC.

For example, when Bob—a Facebook user living in London—refreshes his news feed, the front-end Javascript (JS) application in his browser sends an asynchronous (AJAX) user request to Facebook's European DC in Sweden. The latest posts made by Bob's friends are collected from the replicated data at this DC and returned to the front-end application with low network latency. If, however, Bob decides to post a status update of his own, the front-end application sends a user request to the master, located at Facebook's DC in California. Once written to the appropriate partition on a MySQL instance, a response is returned to Bob.

A full replication strategy for online services such as Facebook and Twitter has a number of disadvantages. In such a strategy all write requests must be directed to a single chosen DC, resulting in poor user response times for users that are located geographically far from

[8] Gizzard, `https://github.com/twitter/gizzard`, last accessed: 13/2/2012

[9] Big Data in Real-Time at Twitter, `http://www.slideshare.net/nkallen/q-con-3770885`, last accessed: 16/7/2013

it. This is due to the increased length of the network path between user and the DC. As a result of this, the throughput of write requests is also affected and depends on the load of this single DC.

Full replication suffers from scalability issues because the replication factor is determined by the number of DCs: each DC requires an entire copy of the data. This prevents global online services from establishing DCs with smaller footprints in certain geographic regions, hosting only a small but relevant subset of the application data for nearby users.

Finally, full replication also raises privacy and security concerns: a DC in a given jurisdiction puts all data under local legislation. For example, Facebook struggles to establish a DC presence in emerging markets due to the legal implications of exposing their North American and European user data.[10] We discuss more subtle issues arising from the data consistency model used with this strategy in Section 2.3.

Content Distribution Networks

Most global online services use some form of a distributed caching solution, in order to improve the response times of global users accessing popular content. Commercial distributed caching services and content distribution networks (CDN) such as Akamai [NSS10], CloudFlare,[11] Amazon CloudFront[12] and Limelight[13] operate by replicating popular content onto multiple geographically dispersed servers in order to provide users with low latency access [PB07, SGD02]. Such CDNs and services serve a large portion of the web traffic we see today with, for example, Akamai reporting to deliver 15-20% of all Web traffic worldwide in 2010 [NSS10].

By storing popular content in the memory of multiple servers, a large fraction of requests issued by users can be handled with low latency. CDNs are therefore critical to the scalability of online services that serve highly popular content, such as viral posts that propagate through a social network or scripts and images for the front-end web application loaded by all users. Distributed caching, however, also has limitations:

A caching solution requires high temporal locality in the read request workload. Unpopular or "tail" content is by definition requested less often and therefore unlikely to be stored on the servers of a CDN [And08, CKR07]. Figure 2.5 illustrates tail content with a typical popularity distribution of data items. The figure shows a few data items that are highly popular, also known as the *head* section of the curve. The majority of the data items are however rarely requested and this is known as the long *tail* section. Tail content is commonly found in online services, such as social networks, where the majority of content is personally curated according to your network of friends or followers.

[10] Will Facebook Friend China?, `http://newamerica.net/node/41985`, last accessed: 10/5/2013
[11] CloudFlare, `https://www.cloudflare.com/`, last accessed: 10/5/2013
[12] Amazon CloudFront, `http://aws.amazon.com/cloudfront`, last accessed: 10/5/2013
[13] Limelight, `http://www.limelight.com/`, last accessed: 10/5/2013

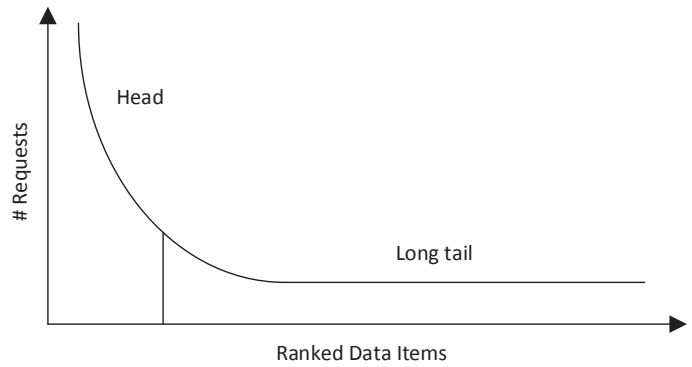


Figure 2.5: Popularity distribution illustrating head and long tail content

Online services such as Facebook and Twitter provide users with a personalised experience. Each user is not only given a different news feed but the content on each news feed also dynamically changes. To generate these feeds, user requests must be directed to the service's application logic in order to obtain access to the stored *original data*—stored data that has not been cached or replicated. This access is required for real-time search and dynamic computation on the latest data, such as calculating the number of comments made to a posted photo on an online social network. User requests must therefore be handled by the application's infrastructure i.e. the origin server, because a CDN is unable to provide such functionality.

2.3 Distributed data management

Database management systems (DBMS) have continuously evolved since their inception in the 1960s [WD06, RG00]. The initial focus of DBMSs was to manage data stored on a single machine. This focus soon shifted to storing data on multiple machines, creating a demand for distributed database management systems (DDBMS). In this section, we discuss two of the most widely used types of database management systems, namely relational database management systems (RDBMS) and NoSQL data stores. Following this, we introduce three key concepts used in DDBMSs and discuss their relevance to latency.

RDBMSs such as Oracle Database,[14] MySQL [RT04] and Microsoft SQL Server, were introduced in the 1970s and became the de-facto standard for storing and managing data by the 1980s. They became widely adopted due to the limitations of previous databases [WD06], their maturity and the large number of supporting tools available [SF12].

RDBMSs operate a relational data model [RG00, GMDW09] consisting of two-dimensional tables called *relations*, each with rows, also known as *tuples*, and columns, known as *attributes*. Data is stored in relations according to a well-define and enforced *schema*. Accessing data in a relational data store is typically achieved with Structured Query Language (SQL), a standardised query language that is flexible and able to support a variety of operations. A key

[14] Oracle Database, `http://www.oracle.com/`, last accessed: 12/05/2013

feature of relational data stores is their support for *transactions* [Gra78, BG80, Ree78, OV11], made possible because of their ACID properties (Atomicity, Consistency, Isolation and Durability).

The Web 2.0 boom brought about a paradigm shift in the design of database management systems. After failing to vertically scale traditional RDBMSs to support large workloads—due to the continuous improvement of hardware in machines either reaching its limit or becoming too costly—, online services considered other techniques. This is why NoSQL (referred to as 'Not Relational' or 'Not Only SQL') data stores were developed.

Although there are many definitions for NoSQL databases, it is easiest to describe them by highlighting their main difference to RDBMSs: NoSQL databases are non-relational and favour *availability* over consistency [Bre00]. In other words, NoSQL moves away from the ACID transactional properties of traditional RDBMSs and towards BASE properties [Pri08].

In favouring availability over consistency, NoSQL data stores are able to scale horizontally. Data can, therefore, be replicated and partitioned across multiple machines in a *shared nothing* architecture. This can drastically improve read and write request throughput in addition to reducing latencies [Cat11]. Furthermore, this provides the opportunity to perform data partitioning and placement across a multi-site wide-area networked infrastructure (see Sections 2.3.1 and 2.4).

Existing NoSQL data stores can be divided into four types: (1) *column-based* stores such as Apache Cassandra [LM10]; (2) *document-based* stores such as Apache CouchDB[15] and Apache Solr [SP11]; (3) *key-value* stores such as Dynamo [DHJ+07] and Memcached [Fit04]; and (4) *graph-based* stores such as Neo4j.[16] Each of these systems have their own properties and distinct trade-offs, and are therefore used for specific applications.

In addition, a new class has emerged from the research community: hybrid data stores mix features found in both RDBMSs and NoSQL data stores. These include systems such as Megastore [BBCF11], a highly available data store with ACID semantics, Google Spanner [CDE+12], a globally distributed database that is semi-relational and provides externally consistent global transactions with their novel global clock synchronization mechanism TrueTime, and Shark [XRZ+13], a distributed shared memory abstraction with SQL functionality. Many of these new systems attempt to push the boundaries on what is possible by using innovative technologies.

In the following sections, we highlight three key concepts used in distributed database management: **data partitioning** and **replication**. We describe these concepts because of their relevance to the data discovery and placement problem of reducing latency.

[15] Apache CouchDB, `http://couchdb.apache.org/`, last accessed: 4/05/2013
[16] Neo4j, `www.neo4j.org/`, last accessed: 4/05/2013

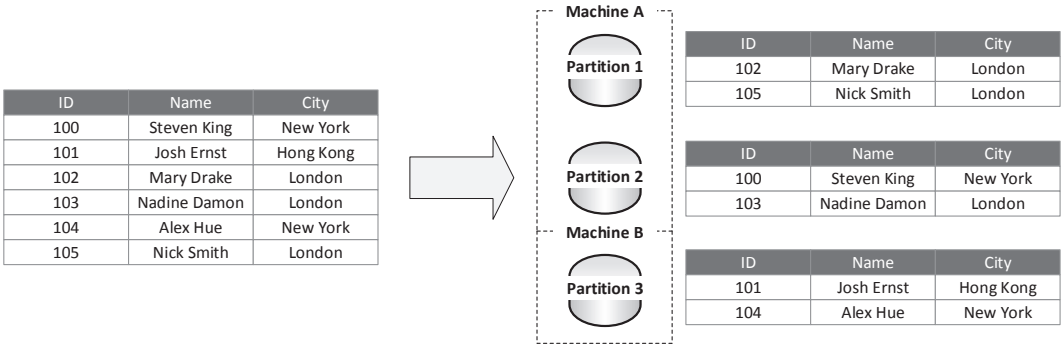


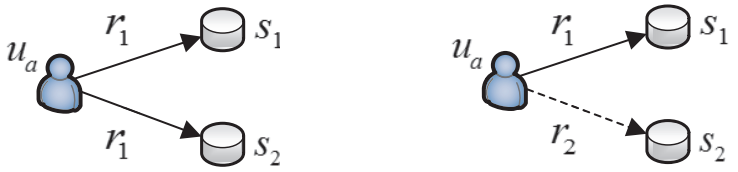
Figure 2.6: Horizontal data partitioning example resulting in three partitions distributed across machines A and B

2.3.1 Data partitioning

As mentioned, a key feature of NoSQL data stores is their shared-nothing infrastructure [Sto86] to achieve horizontal scalability. Database partitioning, sharding or declustering [GDQ92], is a technique that horizontally partitions table rows or records forming multiple *partitions* [RG00]. Traditional strategies perform partitioning according to a given *partitioning key*, a data attribute such as a unique identifier or creation timestamp. Figure 2.6 illustrates the data partitioning process for a table initially consisting of six rows and three columns ("ID", "Name" and "City") resulting in three partitions consisting of two rows each. In this example the "ID" attribute is the key and the strategy is hash-based, which we describe in Section 2.4.1 along with various other partitioning schemes. While data partitioning typically involves the distribution of partitions across independent machines, the figure shows how multiple partitions can be stored on the same single machine: two partitions on machine A and one partition on machine B.

Data partitions provides a number of benefits. Partitioning across multiple independent machines increases the performance of requests due to higher parallelism. While the read request throughput of a service can be improved through replication, data partitioning improves both read and write request throughput. Smaller partitions of data also achieve higher query performance because a smaller working set size is more likely to fit into memory. Furthermore, it enables an infrastructure to be horizontally scaled: expanding its storage capacity on-the-fly by adding additional commodity servers. Lastly, data partitioning provides increases availability as a failure on one machine only takes a small fraction of the data offline. Partitioning strategies are commonly augmented with a redundancy mechanism, replicating partitions to avoid any single points of failure. We discuss replication further in Section 2.3.2.

The partitioning or sharding of data allows for the flexible placement of data (or partitions) across multiple independent machines. These machines can either be arranged into a local-area network (LAN) such as in a single site or data centre, or into a wide-area network (WAN) such as across multiple sites. The majority of schemes are designed for LANs and therefore focus on the partitioning of data rather than the placement of partitions. This is because the placement of partitions within a LAN has a negligible affect on latency.



(a) Low partition isolation (b) High partition isolation

Figure 2.7: Low and high partition isolation illustrated with user u_a, partitions s_1 and s_2, and requests r_1 and r_2

The goal of a partitioning scheme is to increase the parallelism of the infrastructure and thus throughput and latency. This can be achieved by maximising two properties [SC11, Sto86]:

1. *Load balancing.* Balancing the request workload across partitions to ensure each machine handles an equal fraction of the workload.
2. *Partition isolation.* Partition or shard isolation involves reducing the number of multi-get requests which require multiple partitions to be satisfied i.e. *multi-partition* requests. A data partitioning with no multi-partition requests for a given workload is defined as being *locally sufficient* [WK83].

Figure 2.7 illustrates partition isolation with user u_a accessing data stored within partitions s_1 and s_2. Figure 2.7(a) shows user u_a performing a request r_1 which results in both partitions accessed. This scenario has low partition isolation. Figure 2.7(b) shows user u_a performing two different requests r_1 and r_2, each accessing partition s_1 and s_2, respectively. In this scenario both requests are handled by a single partition and therefore has high partition isolation due to it being locally sufficient.

Data partitioning does, however, have its negatives. While providing one of these two properties is simple, achieving both simultaneously is a hard problem. This is because the effectiveness of a partitioning scheme is dependant upon its harmony with a service's *workload*: a complex mix of factors and variables governing the interactions between users and data (see Section 3.1). This dependency places a major burden on database administrators (DBA) and application developers to understand the workload and develop a suitable heuristic that can leverage the general trends exhibited.

A further disadvantage to existing data partitioning schemes is their inability to *adapt* with the workload. An effective static partitioning scheme at one point in time, can rapidly provide poor performance at the next when under a dynamic workload. To simplify the task for database administrators, partitioning schemes must also handle partition re-balancing efficiently. As partitions naturally grow in size, they will need to be subdivided. Many existing schemes require re-balancing done manually and incur service down time,[17] both of

[17]Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes, `http://www.25hoursaday.com/weblog/2009/01/16/BuildingScalableDatabasesProsAndConsOfVariousDatabaseShardingSchemes.aspx`, last accessed: 12/05/2013

which are costly to an online service.

2.3.2 Replication

In addition to partitioning, the availability of a distributed data store is increased through the *replication* of data across various locations in case of power-outages, natural disasters, software defects and hardware malfunctions. Furthermore, replication can also improve the performance of a system by increasing read request throughput. Depending on the consistency model adopted and the workload, the write request throughput can also be improved.

Data *replicas*—i.e. copies of the original data—must be kept consistent when handling write requests, defined as *mutual consistency* [OV11]. The mutual consistency of replicas can be achieved with various consistency models. We describe these consistency models from the client perspective, broadly dividing them into three categories [Vog09]:

1. *Strong consistency* models ensure that if a write request to a data item completes, all subsequent requests to this data item will return the latest updated version. Mechanisms for achieving this include the two-phase commit (2PC) protocol [BHG87] and Paxos protocols [PSL80].

2. *Weak consistency* models are a best-effort approach with write requests to a data item not guaranteed to be seen by subsequent requests. The delay between a non-consistent and consistent state is known as the *inconsistency window* or *replication lag*.

3. *Eventual consistency* models, a form of weak consistency, guarantee that if no new write requests are performed on the same data item, *eventually* all subsequent requests will return the latest updated data item. An example of a system adopting such a model is DNS [MD88]: updated entries eventually propagate through the system with cache expiry times.

These consistency models can be used to implement various replication strategies. Depending on the application and its requirements, strategies must balance the trade-off between the mutual consistency of replicas and the latency of the system [Aba12]. Although this trade-off applies to both a LAN and WAN, it is critical in a WAN due to the high latencies between sites. In the context of the research in this thesis, we provide an overview of replication strategies for multi-site wide-area networked infrastructures i.e. *geo-replication*. We outline different strategies below, describing their effect on request latency.

Single site. The simplest of strategies to ensure availability is to add more resources to a single site. This is often termed *bunkering*,[18] because it combines a number of fail-safe

[18]Transactions Across Datacenters (Google), `http://snarfed.org/transactions_across_datacenters_io.html`, last accessed: 14/05/2013

mechanisms such as multiple back-up power supplies and dedicated network links. Replication is performed across multiple *availability zones* in a single site, each with its own set of resources. The disadvantages include: (1) high availability never being fully attainable due to the possibility of localised outages or natural disasters; (2) high latency of both read and write requests when managing a globally distributed user base due to operating in a single geographical location; and (3) scalability issues when it is not possible to provision the infrastructure further due to physical limitations such as space and energy.

Master-slave replication. A single master-slave strategy follows an eventual consistency model (specifically *read-your-writes* consistency [Vog09]). It is implemented in practice by having a *master replica* at one site, which handles all write requests, and multiple *slave replicas* at other sites. Write requests performed on the master are propagated asynchronously to slave replicas, which eventually become consistent. To ensure that users see their own write requests, it is possible to temporarily divert all user read and write requests to the master replica until the replication lag has passed or cache writes at the replicas. As previously mentioned in the discussion of Facebook and Twitter in Section 2.2.2, a disadvantage of a master-slave replication strategy is the high latency of write requests.

Master-master replication. With a multi-master strategy all replicas accept read and write requests at the expense of either weaker consistency guarantees or performance degradation. Under a strong consistency model, the latency for users suffers with write requests propagated synchronously to other masters. This is enforced with a semi-distributed consensus protocol such as two-phase commit (2PC) [PSL80] or a fully distributed consensus protocol such as Paxos [BG81]. Multi-master replication configurations with eventual consistency assume conflicts are rare and thus asynchronously propagate write requests. However, such a configuration requires a conflict detection and resolution strategy, which handles cases where distributed concurrent updates to the same data item occur.

Another major factor that determines the performance of a system is the *number* of replicas that are kept. We outline two strategies below describing their effect on read and write request latency.

Full replication. Providing a replica per site is known as full replication, which is a high resource cost strategy. Each site must be provisioned to support the storage of all data, and network resources must be available to keep multiple replicas consistent. The advantage of full replication is that user requests (i.e. read or write requests depending on the consistency model adopted) can be handled by multiple sites and thus by sites that are closest to users. This improves the network latency of user requests from a global workload i.e. requests issued by users that are globally distributed. Full replication is adopted by Facebook and Twitter, which we describe in Section 2.2.2.

Partial replication. An alternative to full replication is partial replication (or *n-way* replication), whereby the number of replicas across sites is fixed to n. This reduces the amount of resources required per site while still providing high availability. Latency depends on the *placement* of replicas across sites and how users access data. Depending on the consistency

model adopted either read requests or read and write requests can be handled by replicas.

Replication can be performed at different granularities depending on the application scenario and its requirements. For example, replicas can encompass all of the data, individual data items or data partitions. A replication scheme can therefore be combined with a data partitioning strategy to provide configurable availability, performance and resource utilisation to a multi-site wide-area networked infrastructure.

2.4 Data placement

The placement of data in a multi-site wide-area networked infrastructure in a major factor towards the request network latency experienced by users (see Section 2.1). Placing data at a site that is geographically close to the users accessing it reduces the network path length between user and data, and thus improves the network latency. In this section, we discuss the various data placement and partitioning techniques developed by both industry and the research community, dividing the design space into *static* and *dynamic* placement policies. We include data partitioning policies in this section because they are a form of data placement: data items are divided into partitions, which are in turn placed at different machines. The data placement and partitioning policies we outline are either applicable to multiple machines at a site, e.g. such as in a single DC, or to multi-site infrastructures, e.g. across multiple DCs. The latter policies are able to place data across a wide-area network intelligently and are, therefore, known as *geographically-aware* policies.

2.4.1 Static data placement

The static data placement strategies we focus on are partitioning and declustering schemes used in industry [LTN07, GDQ92]. Although most of these schemes are more suited to multiple machines at a single site, some either support or can be adapted to support multi-site infrastructures. As discussed in Section 2.3.1, data partitioning involves the division of data into smaller partitions. Partitions are created according to (1) a given *partitioning key*, i.e. a data attribute such as a unique identifier or creation timestamp, and (2) a strategy. Each strategy has its goal, which may suit a different infrastructure, request workload and partitioning key. We use the data partitioning example in Section 2.3.1, consisting of a table of six rows and three columns: "ID", "Name" and "City", to present four different data partitioning strategies:

Range-based data partitioning. One of the simplest methods of partitioning data is with a range-based scheme. In this strategy, the range of a partitioning key, such as a unique identifier, is divided into the number of partitions that is required. Figure 2.8 shows a range-based partitioning example using the "ID" attribute as the partitioning key with a range between 100 and 105. The range is divided into 3 partitions, with each partition responsible for storing a fraction of values i.e. two data items in this example. Partition 1 stores data

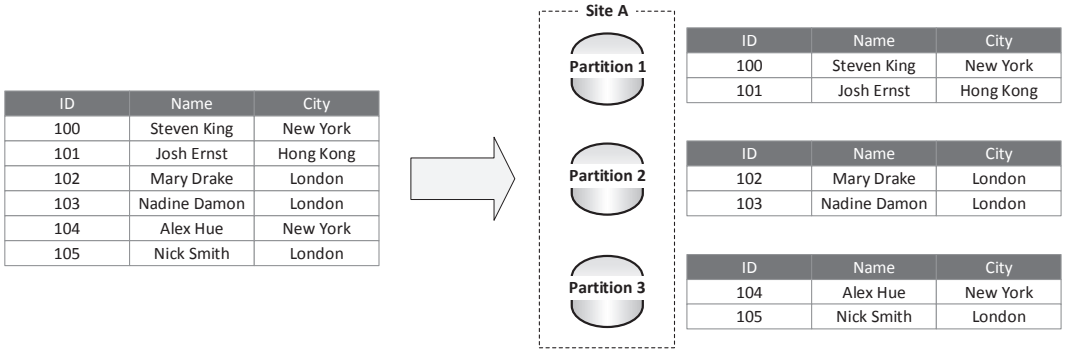


Figure 2.8: Range-based data partitioning example resulting in three partitions (Partition 1–3) at Site A

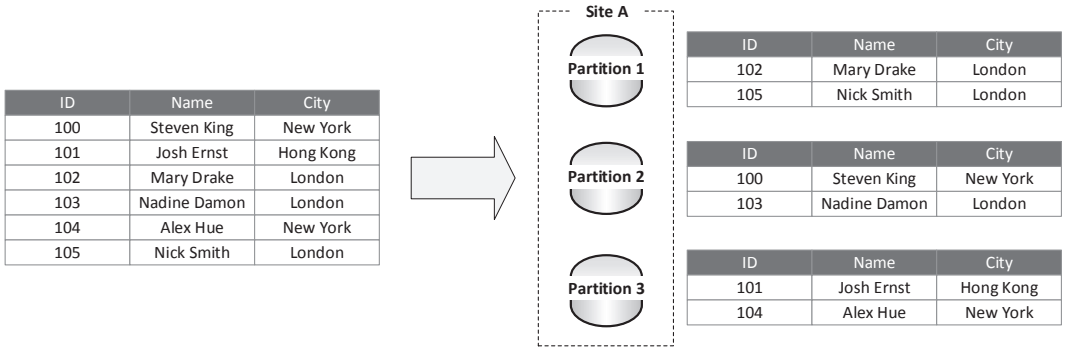


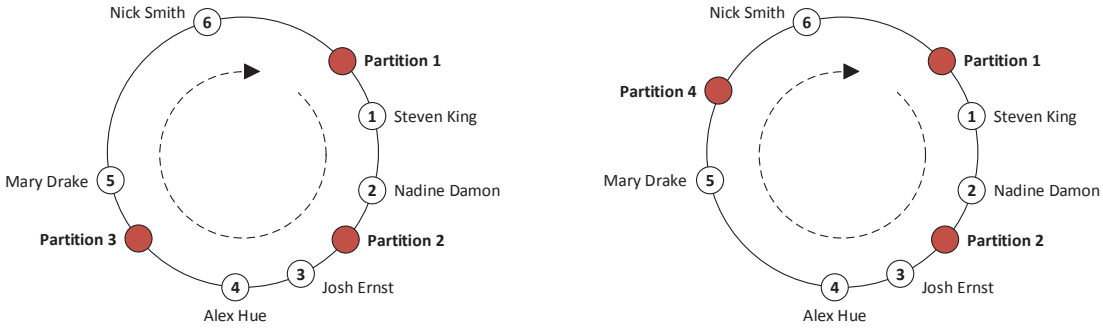
Figure 2.9: Hash-based data partitioning example resulting in three partitions (Partition 1–3) at Site A

items with keys from 100 to 101; Partition 2 from 102 to 103 and so on.

If the partitioning key is requested frequently, this scheme provides high partition isolation for *range requests* i.e. requests that retrieve all of the data items between an upper and lower bound. For example, requesting for users with an "ID" between 100 and 101 results in a single partition participating, Partition 1. This scheme, however, can skew the workload with one partition receiving a larger fraction of requests than another. Range-based partitioning is not a geographically-aware scheme and, therefore, unable to partition data across a multi-site infrastructure to reduce network latencies.

Hash-based data partitioning. A hash-based strategy randomly places data across n partitions. It hashes the partitioning key according to a function and applies the modulus the result, returning a value between 0 and n. Data items are discovered by repeating the above process. Figure 2.9 shows an example of hash-based partitioning using the "ID" attribute as the partitioning key resulting in three partitions, Partition 1–3, at Site A.

The disadvantage to this approach occurs when partitions need to be *re-balanced* e.g. when a partition is removed or inserted. By removing or inserting a partition, the modulus function now returns a different partition for the same key. All of the partitions must, therefore, be re-balanced resulting in $n - 1/n$ data items moved. A solution to this problem is to perform *consistent hashing* [KLL97], treating hash values as a logical "ring" of values with



(a) Initial state

(b) State after removing and inserting partitions

Figure 2.10: Consistent hashing example with a logical "ring" of 1–6 keys partitioned into Partitions 1–4

the largest and smallest hash values wrapping around. In this variation each partition is assigned a random position in the ring, and each data item is placed according to the hash of the partitioning key. Data is placed in the first partition that is clockwise on the logical ring.

Figure 2.10 shows the logical ring for the scenario in Figure 2.9 before and after the removal and insertion of partitions. Figure 2.10(a) shows the logical ring with Partitions 1–3 and data items with keys 1–6. Data items 1 and 2 are stored in Partition 1, data items 3 and 4 in Partition 2, and data items 5 and 6 in Partition 3. Consider what happens if Partition 3 is removed: data items 3 and 4 now belong in Partition 1, and all the other data item mappings are left unchanged. If then another partition is added in the position marked (Partition 4) it will take data items 3–5 leaving only data items 6 belonging to Partition 1. The result of removing Partition 3 and inserting Partition 4 from the ring is shown in Figure 2.10(b).

The benefit of this approach over the standard hash-based partitioning is that the removal or insertion of a partition only affects a fraction of the data items, i.e. the immediate neighbours of a partition on the logical ring. However, the approach is unable to provide partition isolation as it cannot harness the locality in a workload. For example, a range request is likely to require data from multiple data partitions. Furthermore, consistent hashing does not guarantee that partitions are mapped with equal spacing along the logical ring. This results in a skewed load, with some partitions storing more data items, and thus handling more requests, than others. The solution to this is to introduce "virtual partitions", which are hashed to the ring and represent partition replicas. The intuition behind this is an increased number of keys within the ring per partition will balance the load more evenly.

Temporal data partitioning. Temporal data partitioning is a type of range-based data partitioning where the placement of data is done according to a *temporal* partitioning key, such as an insertion timestamp or a date of birth. Time is divided into ranges, such as per-month or per-year, allowing the temporal attribute of a data item to be used to determine its assignment to a partition. Figure 2.11 shows an example of temporal partitioning using

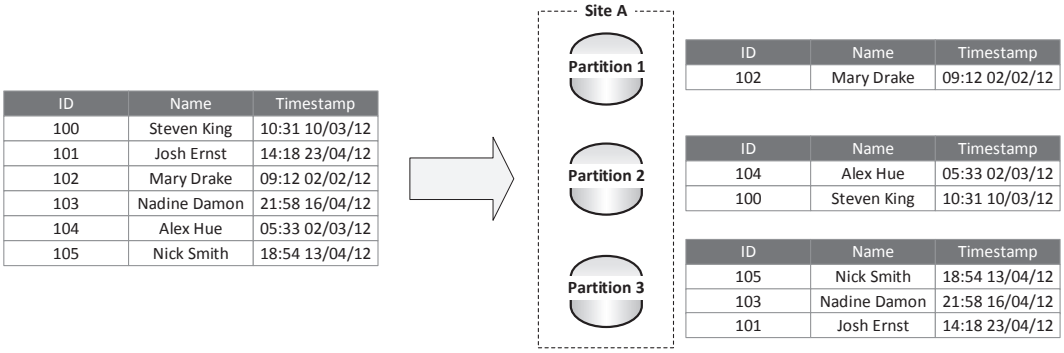


Figure 2.11: Temporal data partitioning example resulting in three partitions (Partition 1–3) at Site A

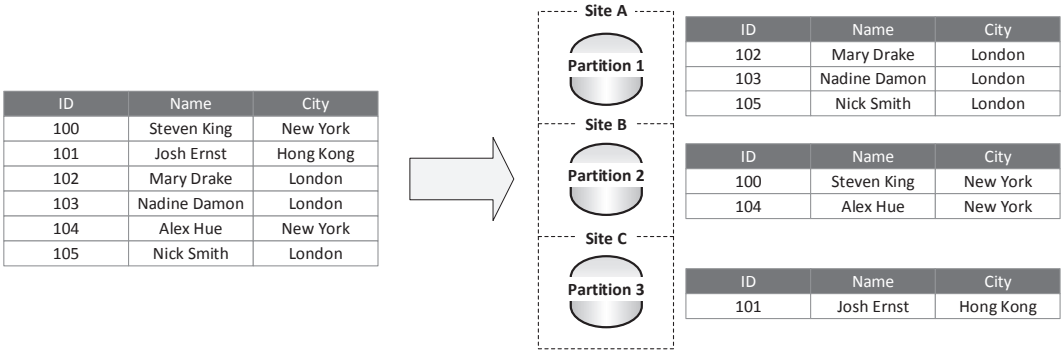


Figure 2.12: Geo-spatial data partitioning example resulting in three partitions (Partition 1–3) distributed across Sites A–C

the insertion "Timestamp" attribute as the partitioning key resulting in three partitions, Partition 1–3, at Site A. The figure shows data items being partitioned according to month. Data items with the latest timestamp, i.e. the month of April, are placed in the most recently inserted partition, Partition 3.

Under a heavily skewed temporal workload where requests are more likely for recently inserted data items, such a strategy achieves high partition isolation but poor load-balancing. The lack of load-balancing is because the majority of requests are for a small fraction of data items, i.e. the latest data items created, which are likely to be stored in the same partition. Twitter is an example of a company that used to adopt a temporal partitioning scheme, as discussed in Section 2.2.2.

Geo-spatial data partitioning. Geo-spatial data partitioning is also an example of range-based data partitioning with the partitioning key related to a geographical location. For example, given a data item that is a user profile containing the user's current place of residence, one could organise partitions according to regions. Users located in the same region are, therefore, co-located on the same partition. Figure 2.12 shows an example of geo-spatial data partitioning using the "City" attribute as the partitioning key resulting in three partitions, Partition 1–3, distributed across Sites A–C. The figure shows each partition handling a particular region: Partition 1 with "London", Partition 2 with "New York" and

Partition 3 with "Hong Kong". Partitions are also stored at sites that are related to these regions. For example, Partition 2 is stored at a DC in New York.

This strategy can provide globally distributed users with low network latencies if, for example, users frequently request for data items which are stored in their same geographical region i.e. high geo-spatial locality in the workload. In such a scenario, geo-spatial data partitioning provides partition isolation (see Section 2.3.1) with requests divided by the region they originate from. An example application scenario with a high geo-spatial locality workload is an online social network (OSN), which stores the profiles of users and their locations (see Section 3.1.4). The strategy is applicable across sites and, thus, is a *geographically-aware* strategy.

A disadvantage of geo-spatial data partitioning is that it assumes that users are uniformly spread across regional boundaries. Any skew in the distribution of users results in poor load-balancing *between* sites, due to some sites storing more data items—and thus handling more requests—than other sites. A further disadvantage is that the strategy is only as good as the accuracy of the attribute representing the region and whether it is the actual geographical location of users.

Although static partitioning has become the industry standard for scaling distributed data stores, it also has its drawbacks. Designing a partitioning policy that has both partition isolation and balances load is challenging. One must identify the access patterns of users that are completely disjoint and separating these data items—i.e. load balancing—, whilst ensuring the data items these users do request are kept together—i.e. partition isolation. We discuss these properties further in Section 2.3.1.

The main disadvantage of *any* static partitioning policy is that placements become outdated when subjected to a dynamic workload. For example, the data attribute used to perform geo-spatial data partitioning may start by accurately depicting the geographical locations of users, but quickly become outdated. Users may travel frequently, move permanently or even lie about their location (see Section 3.1). An example of such a scenario is Facebook, which we described previously in Section 2.2.2. Static policies are therefore unable to adapt according to changes in the workload, as re-partitioning is both costly and time consuming.

2.4.2 Dynamic data placement

Dynamic data placement and partitioning strategies differs to static ones in that their computed configurations—i.e. placement and partitioning of data—can easily be adapted. There has been little research on dynamic strategies across multiple sites. There has, however, been more work which focusses on a single site. In this section, we discuss dynamic strategies that are applicable to a single site and those applicable to multiple sites i.e. geographically-aware strategies. All of the strategies presented touch on central themes of our work: adaptiveness and workload awareness.

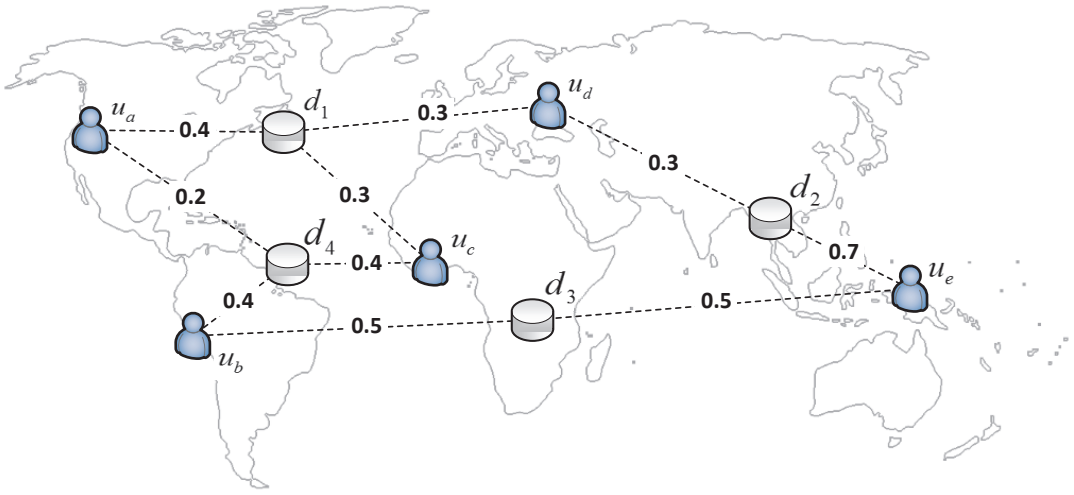
Multi-site systems

Volley. Agarwal et al. present Volley [ADJS10], a dynamic data placement system for geographically distributed multi-site deployments. The motivation behind Volley is to reduce the user response times of participating applications by periodically re-computing the placement of data across a multi-DC deployment. To utilise Volley, applications submit their request logs to a distributed storage system. Request logs are periodically analysed to determine the user-data and data-data interdependencies that occur in the workload (see Section 3.1.5). Volley performs the analysis at a centralised site. Once complete, an optimised placement of data items is determined along with the required data migrations.

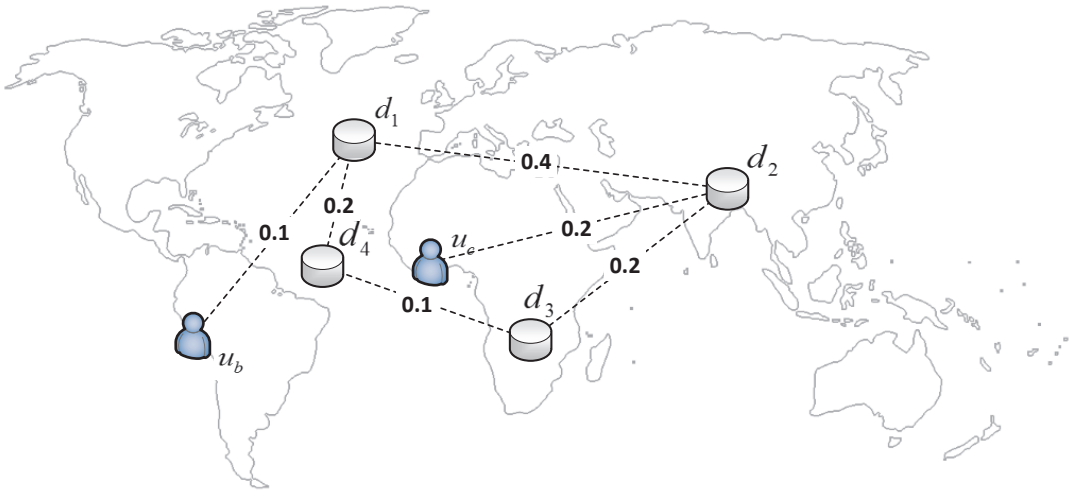
The analysis algorithm runs in three phases, which we outline using Figure 2.13 illustrating an infrastructure with users u_a–u_e, data items d_1–d_4 and sites A–E. In these three phases: (1) it geo-locates users according to their IP addresses and computes geographical coordinates for data items that are *directly* requested by users using the weighted spherical mean calculation. Figure 2.13(a) shows data items being geographically positioned according to the locations of users and their weightings; (2) it iteratively improves the geographical coordinates of data items by considering their communication with other data items and users. Volley is able to capture the data interdependencies found in, for example, publish/subscribe systems [EFGK03]. It achieves this using a weighted spring model, pulling data items and users together that communicate regularly. Figure 2.13(b) shows the geographic locations of data items being adjusted according to their dependencies to other data items and users; and (3) it iteratively collapses the data items and their geographical locations to DCs i.e. find the closest available DC to the ideal location of data items that is not over capacity. Figure 2.13(c) shows the steps taken—shown as encircled numbers—to map data items to sites according to their distance. For this example, we have set the capacity of sites to one data item, and thus why data item d_3 is placed in site E rather than its closest site, site D. Upon completion, Volley has a new data placement configuration, which is compared with the current configuration. The set difference is computed to generate migration plans, which are executed by an application-specific migration mechanism.

Agarwal et al. evaluate Volley using real traces from Microsoft Live Mesh, a service providing users with communication and collaboration features such as remote access and file-sharing/synchronisation. They compare a number of different strategies: (1) single site, storing all the data at a single site; (2) random or hash-based placement (see Section 2.4.1); (3) primary requester, placing data at the site closest to the user accessing it the most; and (4) Volley. The evaluation is performed with 12 globally distributed DCs, and the results show Volley can provide a 30% reduction for the 75th percentile of latency compared to the primary requester placement strategy when using a pre-computed latency model. Interestingly, under a live deployment of the system, these numbers reduce to less than 6% improvement for the same percentile.

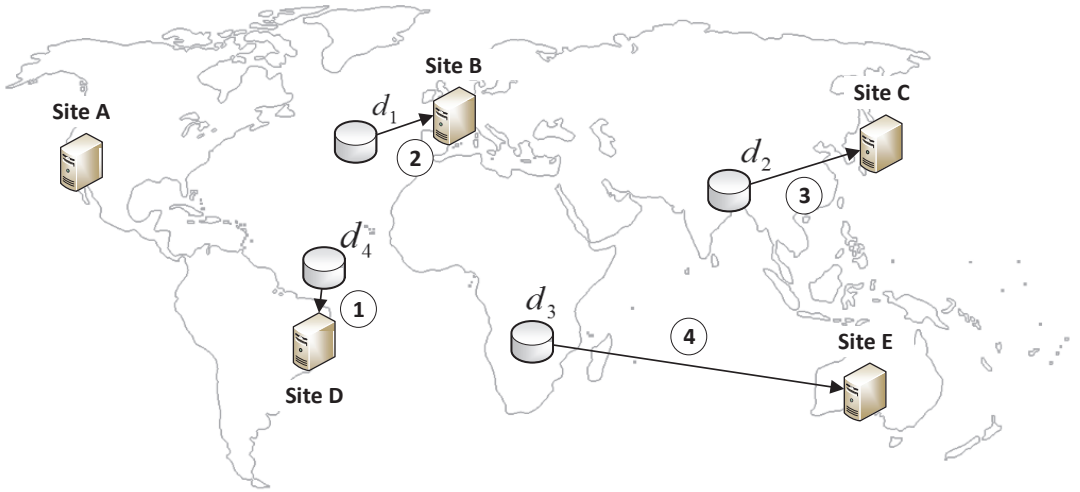
There are multiple disadvantages to Volley. It must periodically aggregate distributed sets of request logs at a centralised site where it requires a large scale distributed execution



(a) Phase 1: calculate coordinates of data items based on users



(b) Phase 2: improve data item coordinates with data interdependencies and other users



(c) Phase 3: iteratively map data items to sites

Figure 2.13: Example infrastructure consisting of users u_a–u_e, data items d_1–d_4 and sites A–E, showing the three phases of the Volley algorithm

infrastructure. This is because of the quantity of data and the algorithm adopted. Volley relies on SCOPE [CJLR08] to perform the analysis, which is a MapReduce-like [DG08] system. A month's worth of request logs for a large distributed application requires over 450 CPU hours to process, rendering Volley unable to adapt to frequent and sudden shifts in the workload. Depending on the online service, such shifts can occur within a matter of minutes, hours and days, not weeks or months as assumed by the authors (see Section 3.1). This problem is further accentuated by the fact that Volley cannot reuse previous results to increase the algorithm's efficiency and improve the execution time of subsequent runs.

When adopting a data placement strategy, it is important to identify how data is discovered. Unfortunately Agarwal et al. do not present details on how this is achieve in Volley. We therefore assume that a forwarding table (see Section 5.4.2) is constructed and disseminated across sites after each iteration of the algorithm. This requires additional network traffic and increases the time required for Volley to update its data placements.

Finally, Volley's results depend on its latency model, i.e. how it estimates the network latencies between DCs and between DCs and users. The authors assume that the model is passed as to Volley as parameter and that it is relatively static i.e. the model has no major changes. However, this may hinder the performance of the system, especially taking into consideration the mobility of users and the increase of global network traffic.

Single site systems

Next we describe systems that perform dynamic data partitioning across multiple machines in a single site. These strategies aim to balance load and improve partition or shard isolation (see Section 2.3.1) dynamically and in line with the workload. As a result, these approaches must understand the interdependencies between data items according to the properties of the application, data or workload.

A disadvantage to all of these techniques is that they are designed for a single site deployment and, therefore, rely on centralised algorithms to perform partitioning. Furthermore, due to the single DC environment, placement of partitions is decided arbitrarily between machines. They therefore have no support for geographical-aware placement of data and hence unable to leverage a multi-site deployment to reduce user response times.

SPAR. Pujol et al. introduce SPAR [PES+10], an incremental partitioning and replication middleware system for online social networks (OSNs). Its goal is to increase the partition isolation during the execution of typical multi-get requests in a single DC. It approaches the problem by: (1) analysing the social graph of a service; (2) partitioning it dynamically—i.e. with each modification to the graph—to ensure tight communities are preserved as much as possible; and (3) minimising the number of partition *replicas* needed to co-locate all of a user's direct social connections—i.e. *one-hop* nodes in the social graph—on the same machine. With a good partitioning and a large enough number of replicas, requests can be satisfied with high throughput and low user response times. SPAR's partitioning scheme is

dynamic: it can decide on the placement of newly created graph nodes at runtime.

SPAR has a number of disadvantages: it focusses on social graphs, which makes it an application-specific solution for OSN applications. It is also unable to support social networks, which have more complex social graphs. An example of a service with such a social graph is Facebook, with vertices and edges used to represent more than social connections between users. Pujol et al. also assume that all operations on the social graph are one-hop and therefore only optimise for the retrieval of *direct* friendships (see Section 3.1.4). No optimisation is performed for users retrieving data that has been created by a "friend-of-a-friend" or "stranger". More importantly, SPAR assumes that all social connections are weighted equally. SPAR may therefore optimise for requests that rarely occur as the actual workload of the OSN is not analysed (see Section 3.1.4).

SCHISM. Curino et al. describe an offline workload-aware database partitioning and replication mechanism called SCHISM [CJZM10], which is designed to reduce the number of distributed transactions in On-Line Transaction Processing (OLTP) workloads. It is described as a general approach for any application with *transactional* workloads [Gra78,BG80,Ree78, OV11]. The approach relies on constructing a graph of data items (or tuples) with edges representing an access on both of them. The graph is partitioned using METIS [KK98] into k balanced partitions, which minimise the amount of transaction traffic.

Although SCHISM relies on query traces to ensure that partitioning and placement are accurate, i.e. reflects the actual workload, its execution is performed offline, non-incrementally and is centralised. Re-partitioning according to the latest query traces requires: the recomputation of the entire graph, its partitioning and its re-distribution. Furthermore, with enough elapsed time, the workload may change considerably. As a consequence, the new partitioning may be substantially different from the last, requiring bulk data migrations. This is a disruptive process when managing data on a live large-scale deployment. Another disadvantage of SCHISM is its sole focus on transactions, disregarding non-relational workloads found in, for example, NoSQL data stores. Furthermore it only focusses on transactional *write requests* and does not optimise for read request workloads, which make up the majority of requests in an online service.

2.5 Data discovery

As we have mentioned previously in Section 2.1, there is a relationship between data placement and data discovery in multi-site wide-area networked infrastructures. When adopting a sophisticated placement strategy, the strategy for discovering data can be rudimentary such as always forwarding requests to the next closest site; similarly when performing a sophisticated discovery strategy, the placement of data remains important but less critical. In the previous section, we discussed various data placement techniques used to place data throughout multiple distributed sites. In this section, we discuss approaches for discovering data in a multi-site wide-area networked infrastructure, assuming that data is assigned and

fixed to sites.

2.5.1 Centralised data discovery

Discovering or searching for data in a distributed environment is a large and broad topic, which has been explored by numerous research communities such as the peer-to-peer (P2P), database, artificial intelligence and semantic web communities. There are two main approaches to discovery, namely **centralised** and **decentralised** [CS02, LCC+02]. A centralised approach involves a single site (defined as an *oracle*) indexing the location and metadata of all data that is distributed among sites. Indexing can be realised according to a *pull model*, where the oracle actively requests the latest data, periodically updating its index; or a *push model*, where sites inform the oracle of updates to their data. Users wishing to discover data submit a request to the oracle. The locations (such as an IP addresses or hostnames) of relevant sites are returned, and the user then contacts the sites directly. Examples of centralised discovery systems include peer-to-peer file-sharing applications, such as Napster[19], and search engines, such as Google and Yahoo!.

Although a centralised approach is simple and effective, it suffers from a number of limitations. It is not scalable when it needs to both index a large number of sites and ensure that their metadata is fresh. In general, there is a delay until the index is up-to-date with the latest metadata, reducing the accuracy of the results (see Section 2.2.1). Availability may also be reduced due to a single point-of-failure. Finally, a centralised approach exhibits higher latencies for globally distributed users because the oracle is positioned at a single geographical location.

2.5.2 Decentralised data discovery

Next we turn to decentralised approaches, which requires for the sites involved to communicate amongst one another. The configuration of sites—or nodes, used interchangeably—and the links between them constitutes a logical topology [LCP05], which operates over a physical network topology such as the Internet. The logical topology is what is used to discover data at sites and is commonly known as an *overlay network* (see Section 2.1). A decentralised data discovery strategy constructs an overlay network in order to reduce the number of links traversed (or *hops*) in discovering the sites that hold the relevant data.

To explore the various decentralised designs and proposals in the literature, we classify overlay networks into two broad groups: *unstructured* and *structured* overlay networks [CS02, LCP05]. Unstructured (and *loosely structured*) overlay networks impose no restriction on how nodes and the application-level links between them are organised. They are therefore able to handle churn (i.e. nodes leaving or joining) and failure. In these overlay networks, data is typically stored on the original site and is not migrated to other sites. Data is searched for using various application-level *forwarding strategies*.

[19] Napster (shutdown 7/2001), `http://www.napster.com/`, last accessed: n/a

Structured overlay networks, such as DHT-based structures (see Section 2.5.4), provide a searchable distributed index abstraction which is based on the data stored and is used to locate nodes. The application-level links between nodes are typically chosen to ensure that requests are handled within a tightly bounded number of hops. In the following sections, we outline the various routing strategies available in unstructured overlay networks and the designs for structured overlay networks. In addition we give an overview of how they operate and the properties that they provide.

2.5.3 Unstructured overlay networks

Unstructured overlay networks are inherently robust to churn in the network, which means that connectivity is largely unaffected by frequent node failures, arrivals and departures. Furthermore, nodes are responsible for storing their own data without the need to propagate it further through indexing or caching. Unstructured overlay networks are therefore capable of processing arbitrarily complex queries, because queries are executed against all of the original data and not an outdated subset.

In an unstructured overlay network, nodes maintain links with only their immediate neighbours in the system, creating a random graph. An example of a system with such a topology is Gnutella [KM02], a first generation peer-to-peer (P2P) file sharing application. Random topologies, however, are intuitively low maintenance networks, requiring less network maintenance traffic between nodes. Other loosely structured topologies exist, such as hierarchical structures composed of clusters and *supernodes* [CS02]. Due to the lack of an index abstraction in these overlay networks and nodes only capable of holding local knowledge, search is performed through mechanisms such as flooding or random walks [CGM02, KGZY02].

Request forwarding in an unstructured overlay network can be categorised as either *blind* or *informed* (heuristic) search [FSB04]. We discuss some of the proposed techniques for each of the two categories in the following sections.

Blind search techniques

Blind techniques perform search on an unstructured overlay network without prior knowledge of the network or past requests. It includes algorithms such as breadth first search (BFS), modified or random breadth first search (RBFS) [KGZY02], iterative deepening (ID) [YGM02] and random walk (RW) [CGM02].

Breadth first search. In a BFS search requests are propagated in a recursive manner with the number of recursions bounded by a *time-to-live* (TTL) parameter. A node begins by querying all of its neighbours i.e. nodes that have a direct application-level link to it. Each neighbour in turn queries all of its neighbours, and so on. This continues until either all nodes have been reached or messages have surpassed their TTL value. Although a simple approach, BFS is highly inefficient due to the large number of messages. It is therefore unable

to scale with a large number of nodes. An example of a system adopting such a strategy, is Gnutella. Later versions of Gnutella use a variation of the BFS algorithm, which is more efficient.

Modified BFS. [KGZY02] is a variation of the BFS algorithm with messages sent to only a configurable fraction (p) of neighbours. Selecting a suitable value for p is essential to get the right trade-off between message efficiency and search effectiveness. With smaller p values, fewer messages are sent at each recursive step resulting in a higher chance of not reaching the desired nodes. A user request could, for example, fail as it was not forwarded to a neighbouring node.

Iterative deepening. Yang et al. propose an iterative deepening technique taken from the artificial intelligence community [YGM02]. Their *expanding rings* technique involves iteratively running BFS at increasing depths. At the end of each iteration, nodes that have received the request store it until the source node propagates a "continue" message to continue onto the next depth iteration. The number of messages that are sent can be substantially larger than that of BFS due to the overhead of having to perform a BFS at each iteration. Furthermore there is no guarantee of a successful discovery unless the maximum search depth is set to the diameter of the network. Finally, network latency is high because of the waiting and "continue" message propagation that occurs between iterations.

Random walk. k-random walks [LCC+02, JW04] involves sending messages to a random k neighbours. Neighbours propagate messages to yet more random neighbours. The requests (or queries) that are forwarded are also known as *walkers* because they follow a single random path through the network. The termination of walkers is determined by their success or failure. Failure can be configured as one of two ways: (1) a walker reaches its TTL; or (2) a walker *checks* frequently with the source node to determine if it should continue walking.

Random walk methods decrease the overhead of search because the number of messages is constant with each iteration. However, the rate of success for requests is low and there is guarantee for data discovery. More recent research has proposed variations on this technique providing stricter search guaranteed and using only local knowledge [ZSS08].

Informed search techniques

Informed search techniques use prior knowledge—such as storing metadata about the location of data—to use as a heuristic when forwarding search requests. Informed search algorithms include directed BFS [YGM02], intelligent search [KGZY02], local indices based search [YGM02] and adaptive probabilistic search [TR03]. This section provides an overview of the mechanisms that this category of search techniques is based on.

Directed BFS. The general concept behind directed BFS [YGM02] is for source nodes to send messages to only a subset of neighbours that have returned results in the past. The subsequent forwarding achieved by these neighbours is performed as in ordinary BFS. Selecting appropriate neighbours is done based on collected statistics. These can be derived

from a variety of heuristics, e.g. the highest number of results returned, the results returned with the fewest hops and the stability of nodes. There are fewer messages sent in this approach than in BFS, while maintaining a high success rate i.e. the number of results returned. However, neighbours of the source must still perform a BFS, resulting in a large number of messages.

Intelligent search. Kalogeraki et al. present a similar approach to that of directed BFS, called intelligent search [KGZY02]. It uses the request *keyword vector*—a vector consisting of the keywords that make up the request or query—to direct requests to a subset of neighbours. Nodes store the vectors of past successful requests along with the neighbour that responded. The algorithm then executes a ranking function to determine which neighbour to forward a request to. Nodes are ranked on a *per-query* basis using the *cosine similarity measure* (discussed further in Section 3.3.2). Neighbours that receive a request and can answer it, return the result immediately. Neighbours that cannot, either stop due to reaching the maximum TTL, or continue forwarding the request to their neighbour with the highest rank.

Intelligent search is a learning algorithm for unstructured overlay networks, providing informed directed search in an otherwise random topology. Although the algorithm can reduce the number of sent messages, the success of requests depends on the stability of the content being stored i.e. if the data is updated often or is dynamic (see Section 3.1.5). When storing dynamic data, for example, the algorithm must unlearn paths that once proved to be successful for certain keywords. Another disadvantage of this approach is its per-query ranking of neighbours, which puts the forwarding decision on the critical path, and thus increases user response times.

Local indices. The intuition behind local indices search [YGM02] is for nodes to maintain indices that contain all the data stored within k hops. Nodes can answer requests with their local indices without forwarding requests. In this strategy, requests are processed at each k hops away from the source. The request is simply forwarded in a BFS manner in-between. The main disadvantage of such a strategy is the need to store another node's data. Not only may this not be possible, such as in our global sensor discovery application scenario, but the stored replica data must be kept consistent with the original data. This increases the network overhead if storing, for example, highly dynamic data.

Adaptive probabilistic search. Tsoumakos et al. propose a probabilistic walking algorithm called APS [TR03]. The source node issues k walkers, which are forwarded to neighbours with the highest probability. Probabilities are calculated based on the success of prior requests. APS has two configurations: *optimistic* and *pessimistic*. In an optimistic approach, nodes perform guesses to direct walkers to their next hop. Upon forwarding of a walker, nodes optimistically update their entry within the index, stating the success of the path. Upon the failure of a walker, a series of messages are sent along the reverse path of the walker to correct the indices of nodes. In a pessimistic approach, this process is reversed, and the indices of nodes are only updated upon the success of a walker.

A major disadvantage to all informed search techniques is their inability to adapt to changes

in either the request workload or the data itself e.g. dynamic data. This is because the statistics that are collected are never discarded, denying an algorithm the ability to unlearn. Learning algorithms are therefore limited when trying to, for example, forget a learnt path in an overlay network.

2.5.4 Structured overlay networks

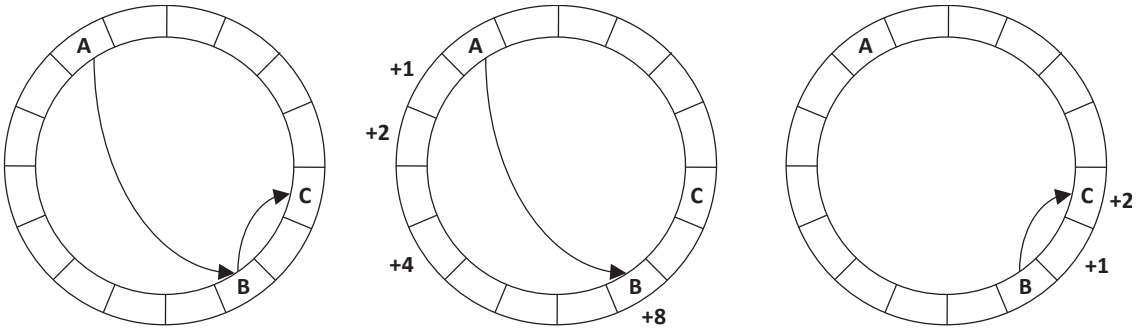
Structured overlay networks organise the topology of the overlay network according to the data that is being stored. In this section, we discuss how a number of these overlay networks are constructed, their strategies for forwarding messages and their advantages and disadvantages.

Distributed hash tables. DHTs [SMK01, RD01, ZHS+04, RFH01] are a commonly used data structure for building scalable distributed applications. They provide a method for distributing key/value pairs (k, v) across nodes, and can retrieve them within a bounded number of hops. Placement of values is determined by the keys themselves, with their hashed value mapping onto an abstract *key space*. DHTs assign partitions of this key space to nodes, which become responsible for storing any values within the partition. Storage and retrieval functionality is provided through a simple interface consisting of *put(k, v)* and v = *get(k)*.

A request for the retrieval of a key is routed across the overlay network, toward the node that stores the values for that key space partition. Nodes in DHTs do not require global knowledge for them to successful forward requests. Instead nodes must only know of a small subset of nodes in the system, which are placed in its *routing table*. Different DHT vary in the structure imposed on the overlay network, but a common theme is for nodes to store links in its routing table to both short (i.e. local) and long distance nodes in the key space. It is because of this that requests are guaranteed to reach their destination within an average of $O(logN)$ hops, where N represents the total number of nodes in the overlay network.

Chord. One of the first DHTs is Chord [SMK01], which operates using a consistent hashing mechanism (see Section 2.4.1). Each node in Chord chooses a random n-bit identifier (*nodeID*), which is typically derived by hashing a fixed unique attribute such as an IP address. NodeIDs are arranged in a ring shaped key space. Each node is responsible for storing the keys that are positioned between itself and the next node clockwise along the key space ring.

Forwarding in Chord is performed using a *finger table*: each node stores the location of other nodes in the ring at increasing distances. Each finger table stores the location of a maximum of m nodes, where $N = 2^m$. The ith entry within the table of node n contains the first node that succeeds n by 2^{i-1} in the key space. This allows a node to locate a key in several network hops using a binary search strategy. Figure 2.14 shows the process of routing a message two hops away from node A to node C. Figure 2.14(a) shows how the path must go via node B, as node A does not have node C in its finger table. Figure 2.14(b) shows how



(a) Two hops from Node A to C (b) First hop from Node A to B (c) Second hop from Node B to C

Figure 2.14: Example of a Chord finger table and routing a message from Node A to C

the first hop is performed using the finger table in node A. As the next entry after node B is at a distance of +16—i.e. beyond node C—node A routes the message to node B at distance +8. Figure 2.14(c) shows the second hop, where node B can route the message directly to node C using the +2 entry in its finger table.

Content Addressable Network. CAN [RFH01] is a DHT, which structures its overlay network into a *hypercube*. The key space is viewed as a d-dimensional Cartesian coordinate space, with one hash function for each dimension. The key space is partitioned into *zones*, which are assigned to nodes. Routing is achieved with nodes keeping links to their direct neighbouring zones in a routing table. Nodes compare the key in the message with neighbouring zones, forwarding the message appropriately. Inserting a key-value is done by hashing each dimension to form a point within the key space, and then routing toward the node that owns this zone. For a node to join, it picks a random point in space, routes a message toward that zone and splits the space (and the maintenance of the zones) into two.

In CAN, each node maintains a link with only $O(d)$ other nodes and is therefore independent of the number of nodes in the system. Routing takes $O(dN^{1/d})$ hops, with more dimensions improving efficiency, availability and fault tolerance of the overlay network. Increasing the number of dimensions does, however, increase the number of neighbours a node has and therefore, the size of the routing table that is stored by nodes.

DHTs impose a logical structuring of nodes that is independent of the underlying physical network topology. Therefore hops traversed on the overlay network may lead to long network paths, resulting in high latency during search. The *stretch* [AMD04] of a system is defined as the ratio between latency in the overlay network and latency in the physical network. Therefore a lower stretch provides lower response time and reduces the unnecessary consumption of network resources. The majority of DHTs have a high stretch, and can therefore be defined as *network-oblivious* overlay networks [PLMS06].

There are however exceptions to this, with some DHTs attempting to improve on latency by storing additional routing information. These are *proximity-aware* DHTs, such as Pastry [RD01] and Tapestry [ZHS+04], which prioritise physically close nodes in their routing

tables, thus improving network latency. Although these techniques help, the underlying topology is still based on a logical identifier space, with nodes and key-value pairs placed at random.

Distributed Sloppy Hash Table. DSHT proposed by Freedman et al. provides better locality properties than traditional DHTs in order to support peer-to-peer CDNs [FM03]. It supports the insertion of *pointers* (or replicas) in order to alleviate hotspots from popular data items and reduce network latency. Values are inserted into the key space using a 'sloppy' technique: pointers to the value are inserted along the lookup path. Retrievals can be achieved earlier in the routing process by encountering a pointer. To reduce the network latency when retrieving values, Freedman et al. proposes to use several DSHTs of increasing network diameter. The diameter is the maximum desired round-trip time between any two nodes contained in it. Routing can be attempted with low diameter DSHTs before using high diameter ones which consist of high network latencies between nodes.

The proposed mechanism favours the lookup of nearby nodes and is therefore inherently *network-aware* [PLMS06]. It achieves this, however, by replicating popular items when inserting them. Freedman et al. has made the design decision to sacrifice consistency in a DHT—by allowing multiple pointers to be inserted—in order to balance load and provide low network latency in a popularity skewed workload. This works well in CDNs but less so in, for example, an ordinary distributed key-value store where consistency matters and the popularity skew in workloads is less accentuated.

All DHTs have a number of limitations. Due to their dependency on an inherently random process—the hashing function—DHTs provide a simple query model where the discovery depends on the exact key i.e. an *exact-match* query model (see Section 2.2.1). Network-oblivious DHTs (or at best proximity-aware) result in high response times when discovering data. Furthermore, DHTs either (1) store a pointer to where the data is stored, in which case the consistency of the pointer, the choice of key and the additional hops required to access the original data, all affect overhead and performance; or (2) store a replica or cached-copy of the data, in which case the data is separated from its producer resulting in further consistency issues.

2.5.5 Semantic-aware overlay networks

Another category of overlay networks, which was first mentioned by Crespo et al. is that of semantic overlay networks (SON) [CGM05], where the structure of nodes and their links is driven by the content that they store. SONs create another layer of abstraction in the overlay network, which groups nodes according to the similarity of their content. Groups usually define the boundary of *topics*—e.g. jazz, country, hip-hop and classical in music genres—either through unsupervised learning or predefined topic names. Nodes can also be a member of more than one group, as the content stored may relate to multiple high-level topics e.g. a track that is fusion between jazz and hip-hop. The motivation for its structure is to be able to discover data more efficiently: narrowing down the number of relevant nodes by

comparing topics with the user request or query. The mechanisms used for grouping nodes in SONs can be divided into three categories [DV10]: clustering, classification or gossiping.

Clustering-based approaches rely on *unsupervised learning* for the formation of node groups within the overlay network. Bawa et al. introduce topic-segmented overlay networks [BMR03] that partition nodes in an *unstructured* overlay network into topic-based groups. Groups are formed by gathering data from each node at one site, clustering the data and then disseminating to nodes. The disadvantages to such a technique is that it is not scalable or a decentralised approach. Furthermore, while the technique is a practical solution for static content—i.e. data that does not change "topic"—it is unable to adapt its structure. Other SON research on unstructured overlay networks include Semantic Peer [PLM08] and DESENT [DNV07].

Structured clustering-based approaches include the work of Tang et al. who propose pSearch [TXD03], a system that combines a locality hashing mechanism called Latent Semantic Indexing (LSI) [DDL+90] with CAN, described in the previous section. The locality hashing function ensures that the indices of semantically related data are maintained by either the same or neighbouring nodes in the system. The system is unable to handle dynamic data, due to the LSI mechanism focussing on one-time insertion rather than frequent updates. Furthermore the system constructs a network-oblivious overlay network, which provides high network latencies during discovery.

Hui et al. propose a protocol for constructing small-world structures, but focus on replication features for handling popularity skewed workloads [HLY06]. Replication requires that data remains static and that a replaces are kept consistent with the original data. This has an added network overhead and can lead to outdated results being returned to users.

Classification-based approaches adopt *supervised learning* techniques, and therefore rely on prior knowledge such as predefined topics. Raftopoulou et al. propose iCluster [RP08], a system where nodes within an unstructured overlay network maintain application-level links to other nodes that are in the same semantic class, in addition to nodes from different classes. Nodes actively seek to link themselves with other nodes that yield a higher similarity. The overlay network is, therefore, a self-organising structure that adapts to changes in the data stored by nodes. Message routing is performed at runtime and involves the comparison between queries and a node's interests i.e. on a *per-query* basis. This is has an added computation and delay in routing queries over the overlay network. Loser et al. propose a taxonomy-based routing protocol and use Chord as a catalogue, storing node classifications [LSZ04]. Both systems have network-oblivious overlay networks that have a high stretch. They therefore provide high network latencies during data discovery.

Gossip-based approaches rely on gossiping protocols [FPRU90,HKMP96,EGKM04] to self-organise nodes within the overlay network and are therefore fully decentralised mechanisms. Aberer et al. propose GridVine [ACM04], a structured SON with an emphasis on separating the logical overlay network and the physical network. It uses a tree structured DHT-like abstraction called P-Grid [ACMD+03] to maintain the mappings from data and metadata

schemas to the overlay network. Data discovery in P-Grid involves routing requests through multiple nodes i.e. hops. Due to the separation between the logical overlay network and physical network, GridVine has a network-oblivious overlay network. Hops on the overlay network are therefore likely to be high network latency links.

A disadvantage of all SON systems is that, by definition, they are unable to support the organisation of data that cannot be categorised. Furthermore not all of the systems described are adaptive and able to alter their structure according to changes in the data stored. Finally, all of these systems construct network-oblivious overlay networks with a high stretch and therefore provides high latencies during discovery.

2.5.6 Network-aware overlay networks

There are a class of overlay networks that are *network-aware* [PLMS06], and build an overlay network that matches the underlying physical network. These overlay networks have low stretch and can route messages with high efficiency. In this section, we discuss an *adaptive* network-aware technique called network coordinates, which alters the structure of its overlay network according to latency measurements taken between nodes.

Network Coordinates. NCs [NZ02] provide each node with a synthetic coordinate of n-dimensions, (typically $n = 2$ or 3) in a shared coordinate space that is based on latency. Each node keeps track of its own coordinate and updates its placement in the space by obtaining round-trip time (RTT) measurements to a subset of other nodes. The subset of nodes consists of both short and long distanced nodes, with distances in the coordinate space providing an estimate of latency. Nodes that are located close to one another in the coordinate space have low latencies between them. Newly taken RTT measurements are used to iteratively update the positioning of nodes in the coordinate space and the links maintained by nodes.

There are two types of network coordinate approaches: landmark-based [NZ01,PCW$^+$03] and simulation-based [DCKM04]. Landmark-based techniques use a fixed number of landmark nodes for other nodes to calculate their latencies to. Simulation-based techniques operate in a decentralised manner. They calculate coordinates by modelling nodes in a physical system, such as the energy in a spring network [DCKM04].

Network coordinates provide an application with a number of advantages [PLMS06]. Measurement overhead is reduced as nodes do not need to measure distances to other nodes to be able to approximate latencies. Furthermore NCs can adapt to changes in the network, with nodes iteratively updating their own coordinate.

Although NCs provide a low latency and efficient overlay network to communicate with neighbours, it is unclear how its overlay network structure can be used for low latency data discovery. While nodes are aware of the network latencies to other nodes, they do not have knowledge of the data that is stored. One possible discovery strategy would be to use a blind search technique such as modified-BFS or iterative deepening (see Section 2.5.3). However, to obtain efficient and low latency data discovery the request workload must have high geo-

spatial locality i.e. users request for data that is located geographically nearby. This avoids user requests being forwarded across a large number of nodes and links.

2.6 Summary

In this chapter, we presented the background material required to position and understand the work in this thesis. We began by providing an in-depth analysis of the problem, outlining the components influencing data access latencies experienced by users in a multi-site wide-area networked infrastructure. We described two factors that determine the network path length between user and data, namely the data placement and the data discovery strategy. While strategies contribute to the final latency of a system, optimising for either one minimises the importance of the other. We use this trade-off to justify one strategy over the other, when there are restrictions in an application scenario.

After that we formulated the problem statement for the two application scenarios explored in this thesis. The first scenario involves a global sensor discovery platform, which interconnects data sources into a unified infrastructure. The motivation is to provide globally distributed users with low latency discovery of fresh sensor metadata. We outlined existing systems that provide sensor data discovery, which include centralised approaches, such as IrisNet, Hourglass and PIER. Following this, we discussed the limitations of these systems, which include inflexible query models, dependence on caching, and organising data sources into static and predefined structures.

The second application scenario discussed, focusses on the distributed management of data for popular online services such as Facebook, Twitter and Google. These services share a common goal: they leverage a multi-DC deployment in order to provide globally distributed users with low latency access to application data. Current multi-DC solutions include master-slave full replication, which is both resource inefficient and provides high write request latencies; and geo-spatial data partitioning, which is both a crude and static partitioning strategy that is unable to adapt to a workload. CDNs are essential for highly popular static content but for services such as Facebook and Twitter much of the data delivered to users is dynamic and personalised.

We next described distributed data management and defined various terms. We outlined the emergence of NoSQL data stores and their differences to traditional RDBMSs. Subsequently we described shared-nothing architectures and the benefits of partitioning data across multiple data stores. Following this, we discussed replication, mutual consistency and the various WAN replication strategies.

Next we described existing data placement strategies, categorising them into static and dynamic approaches. Traditional approaches are comprised of static data partitioning, with policies such as hash-based, temporal and geo-spatial data partitioning. More recent efforts perform dynamic data placement at a single DC and across multiple DCs.

In the final section, we discussed data discovery, focussing on existing strategies for organising overlay networks. We showed the trade-offs between centralised and decentralised approaches in addition to those for structured and unstructured overlay networks. Although unstructured overlay networks provide rich query models, discovery can be inefficient with multiple nodes and links traversed. Structured overlay networks, such as DHTs, can provide bounded discovery but with a limited query model. Semantic-based overlay networks enforce a structure on nodes based on content. Finally network-aware overlay networks are organised to mirror the underlying physical network, but must use inefficient blind search techniques for data discovery.

The majority of existing data placement and discovery strategies are static and oblivious to the workload. These strategies are therefore unable to adapt to the request workload. In the next Chapter we focus on the properties that can be found in online request workloads and how to harness them in workload-aware strategies.

Chapter 3

Workload-Aware Data Correlation

Providing low latency access to data in a multi-site wide-area networked infrastructure is a hard problem. Global online services and applications can achieve this by adopting effective data placement and discovery strategies as described in Section 2.1. These two strategies have an inverse relationship: the adoption of a sophisticated placement strategy only requires a rudimentary discovery strategy, and vice-versa. An effective placement strategy minimises the network latency during data access by placing or migrating data across the available sites. However application scenarios have diverse data management policies, with some unable to decouple data from its originating site. For example, a global sensor discovery platform is unable to separate sensor metadata from its data source or site without requiring additional network communication to keep the metadata fresh. These applications must instead opt for a data discovery strategy that minimises the total network latency when forwarding requests toward data.

The effectiveness of a placement or discovery strategy depends on its awareness of the workload being handled. Online services and applications have increasingly global and dynamic workloads, adding substantial complexity in designing these strategies. Static strategies with workload-specific policies are likely to provide progressively worse latency performance over time due to workloads inevitably evolving. We therefore require a solution that is both general and adaptive to workloads, aiding the development of dynamic data placement and discovery strategies. This chapter introduces *data activity correlation* (DAC), a workload-aware approach, which, when coupled with an appropriate clustering algorithm, co-locates data with similar user access patterns on the same cluster. The DAC approach is applied later in this thesis as a key component in: a dynamic data discovery strategy for our global sensor discovery scenario in Chapter 4; and a dynamic data placement strategy for our global online services scenario in Chapter 5.

We begin this chapter with an analysis of the current workloads found in global online services and applications, outlining the major factors in the interactions between users and data (Section 3.1). This motivates us to argue the case for a dynamic clustering abstraction, describing its high-level requirements in Section 3.2. In Section 3.3 we present DAC and its

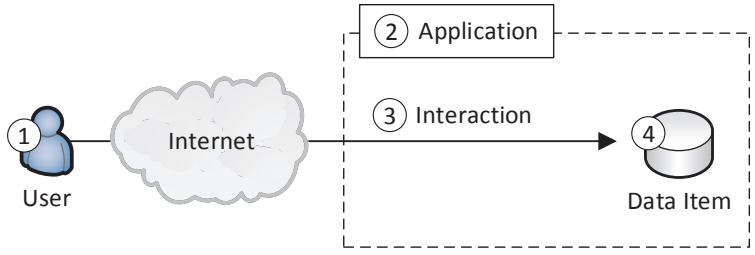


Figure 3.1: Components of the user-data interaction model

components followed by an experimental exploration with synthetic and real-world traces in Section 3.4. We end the chapter with a discussion of DAC, outlining its application to data placement and discovery (Section 3.5) followed by a summary in Section 3.6.

3.1 Global and dynamic workloads

For global online services and applications to provide users with rich functionality, users must be able to access and manipulate stored data. The relationship between users and the data accessed can be complex, comprising of various factors. In this section, we identify the major static and dynamic factors that influence these interactions in modern large-scale online applications. We achieve this through the investigation and review of existing research—performed primarily on online social networks (OSN) due to the availability of these datasets—in addition to our own analysis. A deeper understanding of the workloads of these services allows us to identify the requirements of successful data placement and discovery strategies.

3.1.1 User-data interaction model

To investigate the factors influencing user-data interactions in online applications, we introduce a simplified model comprising of four distinct components: (1) user, (2) application, (3) interaction and (4) data item. These components are illustrated in Figure 3.1: the *user* that begins the process of data access; the *application* that provides the UI and features for the user to interact with; the *interaction* itself, which is delivered over the Internet and turned from a high-level application request into a back-end data store[1] request or query; and the *data item* that is accessed. The following sections discuss each component in further detail, identifying how they can affect the workload.

[1]Back-end data store is defined loosely to represent anything from distributed data stores to a simple XML document.

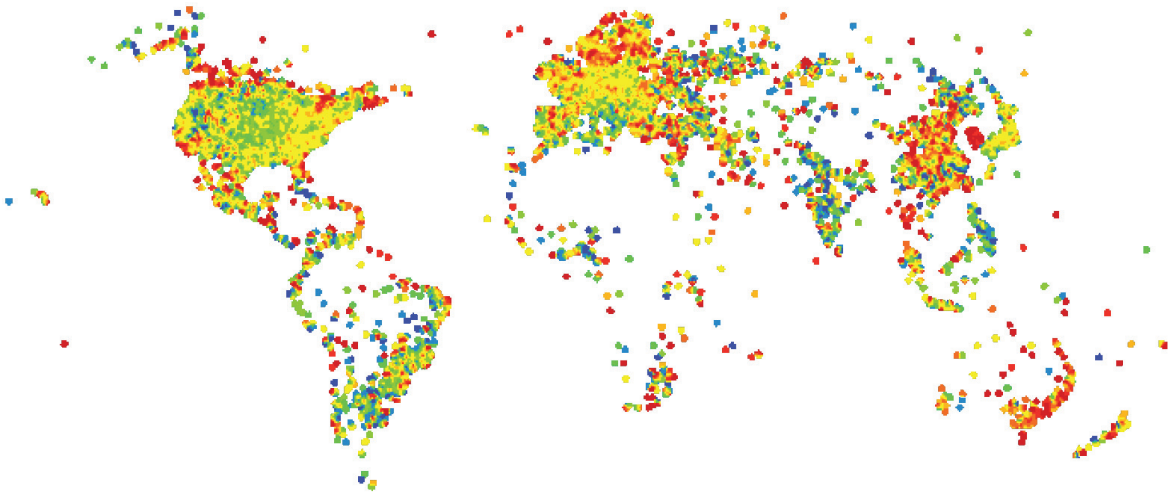


Figure 3.2: Number of users represented with colours in a particular geographical location. Figure obtained from [LH07]

3.1.2 User properties

Each data access or interaction begins with an end-user. Users have a vast number of characteristics including geographical location, social status, social connections and interests. These characteristics not only differ from user to user, they are also evolving with users transitioning between different states. In this section we outline some of the static and dynamic factors of users that are relevant to the problem at hand.

Global distribution. The increase in global Internet adoption has led to a more dispersed and global user base for many online services and applications such as Facebook [Fac12], Twitter and YouTube [KKNG12, BSW12]. Although these user bases are global, their distributions are not uniform. Leskovec et al. investigate a month of communication on the Microsoft Messenger instant-messaging (IM) network [LH07], describing both the global spread and skew in the user base. The results of this investigation can be seen in Figure 3.2, which shows that regions such as the US and Europe are far denser than, for example, South America, Africa and Australia. Further evidence of the skewed geographical distribution of users has been shown in the analysis of Location-Based Social Networks (LBSN) such as Gowalla[2] and Brightkite,[3] illustrating the clustering of users around major cities [CML11].

User mobility. In 2011 the number of smartphones overtook the number of PCs in the world, with this number expected to double by the year 2016 [BGC12]. Facebook mobile usage increased 57% in 2012 with 680 million mobile monthly active users (MAU) as of Q1 2013, accounting for over 64% of the all MAUs on the service [Fac12]. With online services being increasingly accessed from smartphone devices, services have had to deal with varying network path distances between the user and the infrastructure. These distances have been accentuated further by the increase in international air travel year after year [Ama13], with

[2]Gowalla (shutdown 12/2012), `http://gowalla.com/`, last accessed: n/a
[3]Brightkite (shutdown 12/2011), `http://brightkite.com/`, last accessed: n/a

users travelling for short-term, long-term and permanent stays. Agarwal et al. investigate the user mobility patterns of Microsoft Messenger and Mesh. They report that, although most users on their services do not travel, a significant fraction travels far [ADJS10]. They show that 30% of Messenger users travel further than 3,000 miles over a period of a month and 10% beyond 6,000 miles. Further evidence of user mobility has been shown in the analysis of LBSNs describing frequent short-term displacement of users with a probability of 10^{-2} to travel 10 km from their home location and 10^{-5} to travel 1,000 km [CML11].

User diversity and evolution. Users differ from one another with characteristics such as interests, hobbies, education, social connections and careers. Furthermore these users are evolving with these characteristics shifting. For example, graduating from university may affect the social circles a user belongs to or the hobbies they are interested in. Behaviour and user profiling is an active area of research, which attempts to classify users for the purpose of delivering personalised adverts, product recommendation etc [AT99]. Researchers in this field have acknowledged the existence of behavioural shifts in users during both the short and long term, with more recent advancements made towards dynamic profiling techniques [ALA+11].

Discussion. The characteristics of users are both diverse and dynamic. Online services and applications are serving users that are increasingly global and mobile, with smartphones quickly becoming their main device for access. Furthermore users are also diverse and evolutionary, with interests and behaviours shaping their data access patterns. The characteristics of users are therefore a major factor in the user-data interaction model, adding a level of uncertainty in the existence of an interaction between a user and any particular data item stored. Furthermore the network path lengths between users and data items of such interactions are heavily dependent upon the geographical distribution of users in addition to their mobility habits.

3.1.3 Application properties

The applications themselves are a major influencing factor in user-data interactions. The features and components of a UI provides users with a method of interacting with the application, generating an application-level action from each user action. The UI provided to the user is therefore what determines which data items can be accessed. Examples include: an OSN providing the functionality and graphical components for users to submit friend requests; an online calendar application allowing for the creation of events; and search engines providing text-boxes for users to submit keyword-based queries. In this section we outline various application factors that influence this interaction. Furthermore we discuss ways in which applications can evolve, altering the set of data that can be accessed by users.

Application diversity. The broad range of online applications can be divided into three categories based on how restrictive they are toward users accessing data: (1) *restricted* where access to the majority of data on a service is for the most part restricted. For example interactions on Facebook are predominantly restricted to friends; (2) *directed* where users

are directed to accessing specific data by features and the UI.[4] For example Twitter provides users with a list of tweets generated by those they follow. Although users can follow or search for public users and tweets, these are considered additional features; and (3) *open* where the majority of the data stored is accessible to any user. Examples of such services include Google search and Wikipedia.[5] A global sensor discovery platform would provide users open access to data source metadata with the exception of those organisations wishing to explicitly control access to their own data sources.

Software evolution. The software development process for modern online services and applications does not focus on building a one-time deliverable. Services must continuously adapt to changes seen in the market, providing users with new and exciting features to stay competitive. Software development cycles are therefore kept short, with each evolution potentially consisting of new added features, user interface re-designs and application logic alterations. These changes directly affect the users and internal access patterns imposed on stored data. Some of the smallest of features can have large implications for an online service. Viswanath et al. found that over 54% of interactions between Facebook users whom infrequently interact with one another are contributed by their birthday reminder feature [VMCG09]. Larger developmental changes have bigger implications especially when dealing with highly popular online services. For example Facebook have recently moved away from their EdgeRank algorithm [Kin10] for determining which updates get placed in an user's news feed, potentially altering how the application data is gathered and returned on a very large scale.[6]

Apps-on-apps. Further complexity in data access workloads arise when supporting "*apps*" on online services and applications through developer application programming interfaces (API). In these scenarios the control over how data items are accessed is given to third-party developers [GS08]. Many online services are currently supporting a large number apps on their platforms. For example, there are currently more than 10 million apps and websites integrated into the Facebook platform [Fac12]. To achieve the wide-spread adoption of a global sensor discovery platform, it must support the development and integration of apps. This adds an additional layer of complexity to the expected workload.

Discussion. The diversity in applications and their varying affects on user-data interactions suggests that static data placement or discovery strategies will need to be customised for each application scenario. Furthermore the trend for shorter development cycles and developer APIs means these strategies must be revised periodically to ensure they provide the latency they initially set out to give. This calls for both an adaptable strategy—adjusting its configuration according to changes in the application and apps developed on top of it—and a general strategy—applicable to a wide range of applications.

[4] Although most online services and applications can be considered "directed", there is a clear differentiation according to the amount of direction given.

[5] Wikipedia, `http://www.wikipedia.org/`, last accessed: 5/8/2013

[6] A Window into News Feed (Facebook), `https://www.facebook.com/facebookforbusiness/news/News-Feed-FYI-A-Window-Into-News-Feed`, last accessed: 7/8/2013

3.1.4 Interaction properties

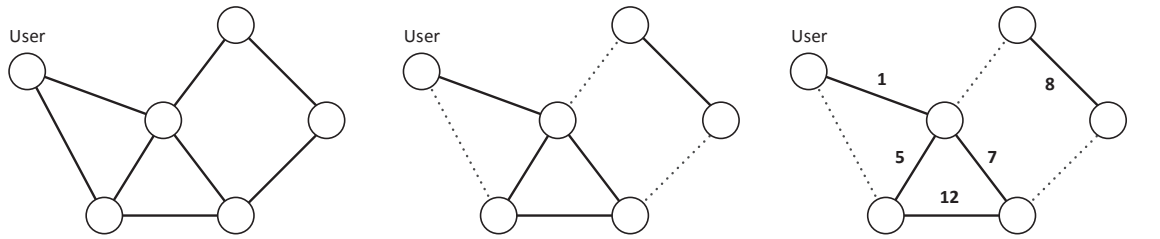
We have discussed the user and application components of our user-data interaction model. This section outlines the properties that can be found in the interactions between users and data, showing the multi-faceted nature of workloads in various global online services and applications. Due to the large amount of research conducted in this area and to ensure we gain a deeper understanding of interactions, we focus on the research conducted on OSNs. Research into OSNs not only gives an insight into the social factors of interactions, but an opportunity to understand other more subtle factors such as trends in topics and the propagation of information through networks.

Popularity (and shifts). Although there have been contradicting reports for the popularity of content in online services [GTC+07], most follow a power-law distribution with many specifically following a Zipf-like distribution [ABCD96, CBC95]. A *Zipf distribution* in the context of content popularity means that a small number of data items are extremely popular, but there is a long "tail" of unpopular data items. Services also experience shifts in content popularity, with the rate of shift varying. Twitter has been shown to be highly volatile with tweets becoming popular through excessive retweeting/mentioning and fading away within a matter of hours [YL11]. Less volatility exists in YouTube video traffic, with large shifts in popularity occurring during a matter of days [BSW12]. Online services and applications with long tails of unpopular content cannot rely on caching solutions for fast access due to the lack of temporal locality in these data items. The churn of user generated content (UGC) and shifts in the popularity of data also add complexity to designing effective data placement and discovery strategies—they must be able to adapt to these changes within a reasonable time frame.

Semantic locality. The data being accessed is typically semantically related to users: for example, OSNs provide data that is *socially* related to users; location based search engines such as Google Maps[7] provide data that is *geographically* related; and online Q&A services and general search engines such as Quora,[8] Google and Bing provide access to data that is *topically* related to the current state-of-mind of users.

Social graphs on OSNs such as Facebook, Twitter and LinkedIn[9] map out the social relationships between users, and therefore dictate the application data that is accessed. Figure 3.3(a) illustrates an example social graph with social connections between six users. However research has shown that users only interact with a fraction of their declared connections, in what has been called the *interaction graph*, a sparser and simpler network of true friendships [WSPZ12]. Huberman et al. show how users' declared social graphs in Twitter differ greatly from their actual interaction graphs [HRW08]. Figure 3.3(b) shows the interaction graph for the social graph in Figure 3.3(a). Data placement and discovery strategies must focus on the actual workload rather than generalised trends of the application. This is im-

[7] Google Maps, `http://maps.google.com/`, last accessed: 20/4/2013
[8] Quora, `http://www.quora.com/`, last accessed: 20/4/2013
[9] LinkedIn, `http://www.linkedin.com/`, last accessed: 20/4/2013



(a) Social graph (b) Interaction graph (c) Interaction graph showing favouritism

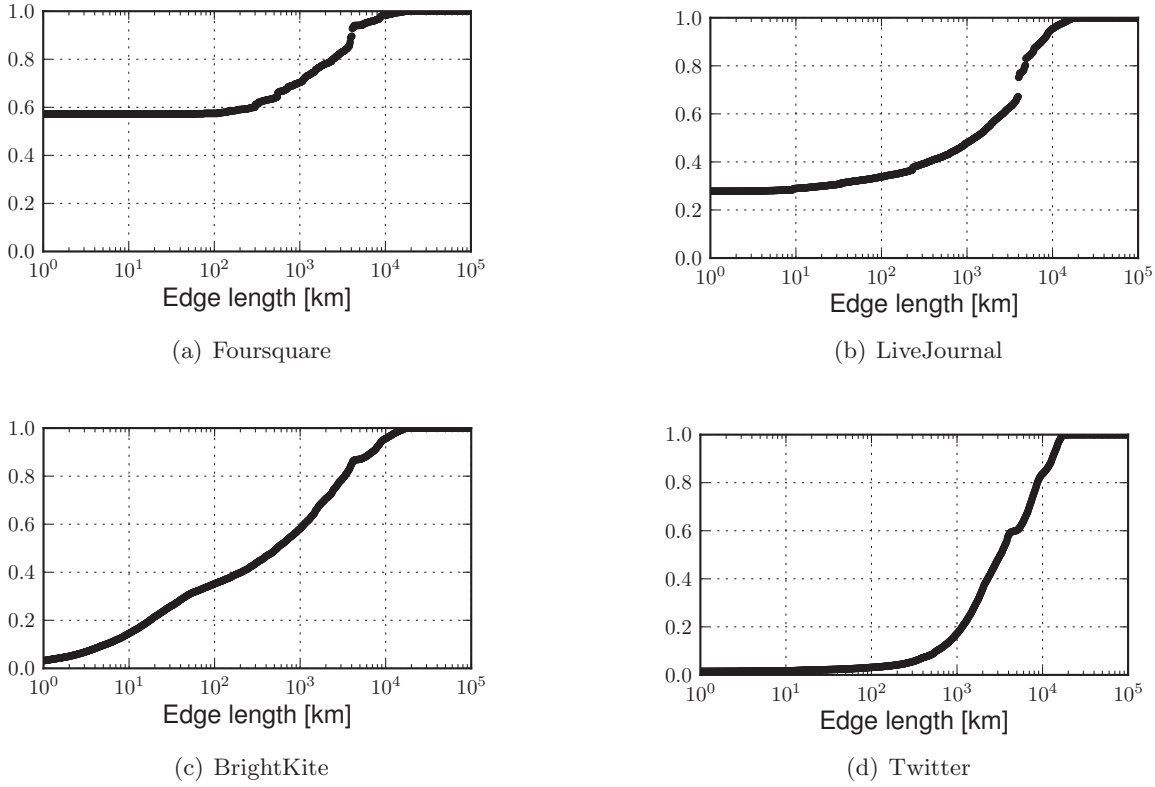
Figure 3.3: Example social graph, interaction graph and interaction graph illustrating favouritism

portant as, for example, the placement of application data according to the declared social graph is an inaccurate heuristic of the workload.

The links in a user's interaction graph—a subset of the full social graph—are also given varying importance by users. Kwak et al. describe how Twitter users exhibit *favouritism* in the interaction graphs, with users only retweeting from a small number of social connections and only a subset of a user's followers retweeting their tweets [KLPM10]. The edge weights or number of interactions between two users in the interaction graph are not uniformly distribution but rather skewed. Figure 3.3(c) illustrates favouritism in the interaction graph with the numbers between social connections stating the number of interactions that have occurred between two users. Wu et al. give an insight into the reasoning for skewed interaction graphs by investigating information cascades on the Twitter network [WHMW11]. In this study, Wu et al. broadly categorise elite users as celebrities, media, organisations and blogs, and show their share and retweet behaviour between one another. Results show that celebrity, media and blog elite users all greatly favour sharing tweets with other elite users in the same categories. The same effect is shown in retweeting amongst blog and media elite users.

Although coarsely described in this study, information cascades in Twitter are driven by the characteristics of users themselves and the semantics of the content they share and retweet. This re-enforces our argument for placement and discovery strategies to focus on the workload rather than structured data such as the social graph. Furthermore it shows that in accurately capturing the true relationships between users—and the data items users generate and access—a strategy can leverage the repeated propagation of information according to these relationships.

Geo-spatial locality. The reported geographical properties between user and data differs widely between services. The analysis of Leskovec et al. on Microsoft Messenger show the general trend for communication amongst users to be localised [LH07], where most of the interaction through shared application data is done by users located in the same region. Brodersen et al. investigate the locality properties of YouTube video traffic and also report high geo-spatial locality, with 50% of videos having more than 70% of their total views in a single country [BSW12]. These services exhibit a clear separation in their workloads based on the geographical distribution of users and the geo-spatial locality of user interests.



(a) Foursquare

(b) LiveJournal

(c) BrightKite

(d) Twitter

Figure 3.4: Cumulative Distribution Function (CDF) of edge lengths in four OSNs. Figures obtained from [SM10]

The relationship between social connections on OSNs and their physical distances has been mixed. Scellato et al. analyse the geographical properties of links found in four OSNs: LBSNs FourSquare and Brightkite, LiveJournal (a blogging social network) and Twitter [SM10]. Figure 3.4 show the distribution of edge lengths in kilometres for these four networks. The distribution of edge lengths in FourSquare, shown in Figure 3.4(a), shows social connections to be highly localised with over 40% of edges shorter than 1 km and an average edge length of just 1,296 km. This is most likely due to FourSquare's support for only 100 global locations during the gathering of this dataset, limiting the possible spread of users. The distribution for the LiveJournal service in Figure 3.4(b) also shows highly localised social connections with 30% of edges shorter than 1 km but a larger average edge length of 2,727 km. Figure 3.4(c) shows how BrightKite has a near uniform distribution of edge lengths ranging from extremely short to 10,000 km. Only 4% of edge lengths are short and the average edge length was reported to be 2,041 km. Edge length distribution in the Twitter social network, shown in Figure 3.4(d), differs from the other OSNs: only 20% of links are shorter than 1,000 km and 10% longer than 10,000 km. The average edge length was also reported to be 5,117 km, much larger then with previous services. The variation of edge length distributions found in different OSNs is high, ranging from highly geographically localised services such as FourSquare and LiveJournal to globally spread services such as Twitter. This variation shows that data access patterns on global online services and applications can have a wide

range of geo-spatial locality properties. We therefore require data placement and discovery strategies that are general, i.e. applicable to multiple application scenarios.

Although the analysis of social graphs provides an insight into the geo-spatial properties of application data access on OSNs, it is the interaction graph that depicts the actual workload of these services. The relation between these two graphs and their geographical properties have been explored by Kaltenbrunner et al. They report that although geographic distance can heavily influence the social connections established by users, its influence on actual social interactions is weak [KSV+12]. Geographical distance has, therefore, little effect on favouritism found in an interaction graph, with long distance connections being as likely to be used as their shorter counterparts. Such an insight reinstates the need for workload-awareness in a strategy, as simply leveraging the properties of the data (in this case the social graph) can lead to inaccurate placement and discovery.

Edge length on an OSN describes the distance over which application data is accessed when an interaction occurs between two users. This is also known as single hop cascading or propagation of information. On many services users can propagate information further by retweeting or sharing, resulting in multi-hop information cascades. Ye et al. study information propagation on Twitter and report that 37.1% of tweets were of 4 or more hops [YW10], resulting in long distance information propagating. This is verified by Kulshresth et al. , reporting that although users are more likely to connect and exchange information with other users from their own country, a third of all links and tweets cross national boundaries [KKNG12]. The assumption that all global online services exhibit high geo-spatial locality in their workloads is therefore incorrect. Instead users on some services access content that has been propagated across long distance links. Strategies assuming high geo-spatial locality risk inaccurate configurations, resulting in high access latencies for users. This reinforces the need for workload-aware strategies that capture the relationships between users and the data they accessed, and thus the geo-spatial locality of the workload.

Real-world dataset analysis. We perform our own investigation into the geo-spatial locality properties of workloads using a real-world query dataset from the AOL search engine [AWBG07].[10] The dataset consists of 20 million web search queries collected from 650,000 users in the US over a three month period (March 2006 to June 2006). To gather the geographical distribution of users who perform such search queries, we use the Google Insights service[11] to calculate the uniformity of users across the 51 states of the US ($s_1 ... s_M$). With a given query, Google Insights provides the normalised *search volume* for each state with an integer from 0 to 100 relative to the highest state, which is always 100. We denote the search volume for query i in US state k as v_{ik} and the total search volume across all states for query i as $V_i = \sum_{k=1}^{M} v_{ik}$.

Our collection process consists of sampling the AOL search dataset and requesting the search

[10] AOL Search Dataset, `http://www.infochimps.com/datasets/aol-search-data`, last accessed: 10/2/2012

[11] Google Insights for Search (shutdown 09/2012), `http://adwords.blogspot.co.uk/2008/08/announcing-google-insights-for-search.html`, last accessed: 20/6/2013

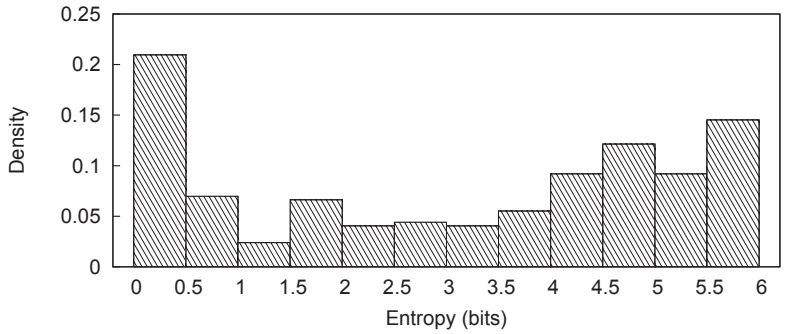


Figure 3.5: Probability Density Function (PDF) of volume entropy for the AOL query dataset

volumes for these queries from Google Insights, resulting in a dataset of 544 queries. Google Insights is unable to provide results for queries that have a low search volume such as those with rare, obscure or specialised keywords. Furthermore the service limits the number of requests that can be performed in a given time period. We adopt a similar methodology as used in past work [BSW12, DG00] to analyse the uniformity in the geographical distribution of users for each query:

$$H_i = -\sum_{k=1}^{M} \frac{v_{ik}}{V_i} log_2 \frac{v_{ik}}{V_i} \tag{3.1}$$

$$H_{min} = 0 \tag{3.2}$$

$$H_{max} = log_2(M) \tag{3.3}$$

We define the *volume entropy* (H_i) for query i in Equation 3.1, which is based on the entropy measure used in information theory [Sha48]. Note that the sum is only run over states for which $v_{ik} \neq 0$. The equation provides a measure of uniformity in the search volume across US states expressed in *bits*. A low volume entropy represents a skewed probability distribution and thus the search volume is localised in few states; conversely a high volume entropy represents an equal probability from any state. Equation 3.2 denotes the minimum number of bits that can be represented i.e. zero, and Equation 3.3 denotes the maximum number of bits i.e. 5.672 for $M = 51$.

Figure 3.5 shows the PDF of volume entropy for the dataset collected. The figure shows 21% of queries with a volume entropy between 0.0 and 0.5 bits, and 41% between 0.0 and 2.0 bits. This indicates that a large fraction of queries are highly localised and requested by users in few states. These queries include keywords with a specific city, tourist attraction, sport team or other geographic related keywords, and tend to be longer queries. For example, queries with 0.0 bits include "atlanta georgia subway", "manufactured homes in maine" and "jewelry miami florida". The figure also shows 14.5% of queries between 5.5 and 5.672 bits (due to H_{max}), and 45% between 4.0 and 5.672 bits. Thus another large fraction of queries

are requested by users spread uniformly across 51 states. These queries general keywords and tend to be one-word or short queries. For example, queries with more than 5.5 bits include “cleaner”, “poultry” and “bankruptcy”.

Our analysis into the geo-spatial locality properties of users using a web search engine shows that users perform queries that are both localised and uniformly distributed. However the analysis does have its limitations. As mentioned previously, Google Insights does not provide results for queries with low search volume. On inspection, the discarded queries tend to be longer and more specialised, and thus likely to also be localised i.e. have low volume entropy. We therefore expect the true proportion of localised queries to be higher than what we have reported in Figure 3.5.

Temporal locality. The workloads of global online services and applications also have temporal properties. Atikoglu et al. perform an analysis of the workload handled by Facebook’s Memcached deployment[12] (a distributed key-value caching store), giving a detailed account of the distribution of requests for the same key (repeated keys) on a number of application domain pools [AXF12]. Their results show that a large majority of keys (45-50%) occur on a very small number of requests (∼1%), indicating a low hit rate on the cache. However as expected, they also have few highly popular keys which are requested millions of times per day.

This demonstrates the long tail distribution of content popularity found in global online services such as Facebook, with many distinct requests handled by their back-end infrastructure. Such a distribution is expected for services which are social and deliver highly personalised and dynamic content to users. Data placement and discovery strategies must therefore be applicable to both popular and tail content, providing low latency access for the entire dataset stored.

Kwak et al. investigate the temporal properties of Twitter and focus on the lifetime of tweets and their propagation. They show that more than 50% of retweets are done within an hour of the original tweet and over 70% within a day [KLPM10]. This illustrates a very short time-lag between tweet propagations in the service, and thus the overall fast paced consumption of information. This churn requires for placement and discovery strategies to adapt to the workload with timeliness, and thus be a dynamic process.

Semantic locality shifts. The evolution of users and their social connections, interests, education and hobbies directly influences the semantic locality of users and data. This is reported in research studying the evolution of OSNs, in which social connections continuously are broken and created over time. Viswanath et al. investigate the interaction graphs for Facebook users, and report that the graph changes 70% of its links during the period of a single month [VMCG09]. Furthermore they report that the interactivity between two users drops drastically over time after the creation of a link. Kwak et al. report on the distribution of the duration of top trending topics on Twitter, with 31% of topics reported as lasting 1 day and only 7% lasting longer than 10 days [KLPM10]. This indicates major and frequent

[12]Memcached, `http://memcached.org/`, last accessed: 14/5/2013

shifts in the interests of its users and reinforces the need for adaptable strategies that are dynamic.

Geo-spatial locality shifts. Geographical locality between users and data on online services and applications has also been reported to shift and evolve. Brodersen et al. report a relationship between the geographical spread of interest in YouTube videos and their popularity. They describe a *spray-and-diffuse* pattern where on average videos first become popular in one region, then move into multiple other regions, before losing popularity with focussed attention from one region again [BSW12]. Intuitively geo-spatial locality shifts on OSNs occur with the evolution of interaction graphs, in addition to shifts in trending topics which are propagated through this interaction graph. This has direct implications when designing data placement and discovery strategies, with static approaches generating configurations that can become outdated. Strategies must therefore look toward the dynamic placement and discovery of data.

Lastly search engine queries have also been shown to exhibit spatial variation. Backstrom et al. propose a model for tracking such variations and evaluate this model with the query logs of Yahoo!,[13] a major search engine. In their evaluation they describe many examples, such as the successful tracking of hurricane Dean according to user-submitted queries [BKKN08]. Further evidence of geo-spatial variations includes Google's global influenza tracking system, also achieved by using query logs [GMP+08]. This reinforces the need for dynamic policies that are able to adapt to the geo-spatial variations found in search engine query workloads.

Discussion. User-data interactions are influenced by a variety of factors, depicting both the presence and properties of an interaction. Popularity shifts in data items are both frequent and short-lasting. Geo-spatial locality depends on the online service, with some exhibiting local interests in data and others more global and geographically dispersed interests. Semantic locality of users on an OSN differs from their claimed social connections. Data is requested by a smaller fraction of connections and potentially by users multiple hops away in the social graph. Furthermore these factors are continuously shifting: geo-spatial locality experiencing a spray-and-diffuse pattern according to the popularity of data and shifts in user interests; and social and interaction graphs of OSN users evolving with friendships made and broken.

For a data placement and discovery strategy to be effective, it must be workload-aware and operate at the *interaction-level* i.e. where individual requests are performed by users to data items. At this level, a strategy can handle arbitrarily complex workloads and thus is able to provide accurately matched configurations. For example, in leveraging the individual interactions on a OSN, a strategy can determine the interaction graph, favouritism and even patterns in the propagation of content. Furthermore, these properties can be determined without being given the social graph. Operating at the interaction-level also ensures the strategy is a general solution that is applicable to multiple application scenarios.

[13] Yahoo! Search, `http://www.yahoo.com/`, last accessed: 20/5/2013

3.1.5 Data item properties

The last component of the interaction model that we investigate is the data item itself, focussing on the properties that influence how users interact with data items.

Dynamic data. A data item itself can also change, affecting how it is accessed by users. This may be due to a user or another entity updating the data item. For example online search engines such as Google provide users with the latest web content according to their submitted queries. When a web page updates its content, a crawler must identify the changes and update its entry in the distributed index. Any changes made to the index such as keyword removal/additions or alterations in their weights, affects how users can access it.

Dynamic data is also present in a global sensor discovery platform as described in Section 2.2.1. Sensors provide a continuous stream of data made accessible to users through their metadata. This metadata is updated dynamically with the latest observations from the stream, enabling users to discover sensors accurately using the freshest metadata. Dynamic metadata can therefore affect the access patterns of such a platform. Intuitively this is because a request for a sensor may match its metadata at one point in time, but fail to match at the next.

Shared data. Modern online services use shared data to enable communication, collaboration and interactivity between users. For example, Facebook users perform write requests on users' walls which are then read by the wall's owner and owner's friends. This added dimension adds complexity in accurately placing and discovering data across sites [ADJS10]. This is because each data item is requested by multiple users—potentially spread globally—with a given configuration affecting multiple network path lengths.

Data interdependencies. Application data in online services can also have interdependencies. A request to update one data item may result in the application logic subsequently updating multiple other data items. This is typical of, for example, publish/subscribe systems [EFGK03] where data published by one user requiring its propagation onto the feeds of multiple other subscribed users [ADJS10]. The implication of this is further complexity in the accurate placement and discovery of data. Strategies must understand the relationships between data items, in addition to those between users and data items. This reinforces the need for a workload-aware strategy that operates at the finer granularity of the interaction-level, and thus is able to encompass all of these relationships.

Discussion. Data is a key component to the user-data interaction model where it can be: dynamically updated through application logic, users or other components such as sensors; shared and accessed by multiple geographically dispersed users; or interconnected with other data through application logic and publish/subscribe systems. An effective data placement and discovery strategy must therefore capture the user-data and data-data relationships found in the workload to provide an informed decision making process. Furthermore it must adapt these relationships when necessary, thus dynamically altering its strategy to provide consistently low data access latencies.

3.2 Workload-aware clustering

In the previous section we discussed the current workloads of global online services and applications, demonstrating the wide range of properties that exist. Furthermore we showed how these properties experience both frequent and radical shifts, affecting the interaction between users and the data stored by these online services and applications. Optimising for low latency data access in a multi-site wide-area networked infrastructure is therefore a difficult task. This calls for an application-agnostic and adaptive mechanism that can be used as a building block for placement and discovery strategies. We argue that *clustering* is the right abstraction for this supporting mechanism.

Data partitioning, clustering and sharding are domain-specific terminologies for the same concept: the division of items according to a policy, metric or measure into n groups. This is a well studied and common technique used to solve problems in multiple disciplines, each with their own metrics and algorithms [JMF99]. Our investigation into clustering is for the purpose of aiding strategies that perform dynamic placement and discovery of data across a multi-site wide-area networked infrastructure. Placement can be seen as the clustering of data and its mapping onto sites, and discovery as the clustering of sites, each with its own data, in a network.

Forming an effective heuristic to partition data or form clusters in an online manner is still an open challenge. We set out four key requirements for a data clustering mechanism, with the goal of reducing user-perceived latency during data access:

1. *Request locality.* We want to maximise the likelihood of each user obtaining their requested data from the same single cluster. This is a more restrictive form of cluster or partition isolation (defined in Section 2.3.1) and ensures that users can be assigned to the most appropriate cluster, which handles requests locally and thus with lower latencies.

2. *Application-agnostic.* The mechanism should provide a generic method for determining appropriate clusters, detecting the relationships between data in a wide range of application scenarios and under a variety of workloads.

3. *Adaptability.* Cluster memberships must be adaptive to encompass the changes that occur in the request workload, and thus the relationships between data. Adaptability should also be performed within a reasonable time-frame to ensure workload shifts are detected.

4. *Access prediction.* An ideal clustering mechanism will predict the future requests of users, which have never requested particular data before. This differs from optimising for repeated requests made by users i.e. temporal locality.

The rest of this chapter introduces data activity correlation (DAC), a workload-aware clustering approach that aims to fulfil the requirements set out above. We use it as a key com-

(a) Low cluster isolation (b) High cluster isolation

Figure 3.6: Cluster isolation illustrated with user u_a, clusters c_1 and c_2, and request r_1

(a) Low request locality (b) High request locality

Figure 3.7: Request locality illustrated with user u_a, clusters c_1 and c_2, and requests r_1 and r_2

ponent in the research and development of various distributed data placement and discovery strategies used in our two application scenarios described in Chapters 4 and 5.

3.3 Data activity correlation

Data activity correlation (DAC) uses the current workload as a heuristic to cluster data, with the goal of providing clusters or partitions with *request locality*. Request locality requires that a clustering: (1) maximises the number of requests that are handled by a single cluster, i.e. that provide *cluster isolation* as described for data partitioning in Section 2.3.1; and (2) increases the likelihood of requests issued by each user to be handled by the same cluster, i.e. provide each user with a preferred cluster that can satisfy all or most of their requests over time. Both requirements ensure that user requests can be handled locally and without the need to propagate requests further.

We illustrate cluster isolation, request locality and their relationship with Figures 3.6 and 3.7, respectively. Figure 3.6 illustrates user u_a requesting for two data items in two different scenarios, one with low and another with high cluster isolation. In Figure 3.6(a), the data items are placed in clusters c_1 and c_2, thus requiring the user request r_1 to be directed to two distinct clusters. The number of multi-cluster requests is large and thus represents low cluster isolation. Figure 3.6(b) shows the case where the data items are co-located in cluster c_1, with user request r_1 directed to a single cluster. The lack of multi-cluster requests means high cluster isolation.

Figure 3.7 illustrates low and high request locality—a more restricted form of cluster isolation—with user u_a requesting two data items at different times. In Figure 3.7(a), these data items

are separated across clusters c_1 and c_2. In this scenario, user u_a first issues request r_1 to cluster c_1 at time t_1, followed by request r_2 to cluster c_2 at time t_2 where $t_1 < t_2$. While the number of multi-cluster requests is zero and thus represents high cluster isolation, the clustering provides low request locality due to the user accessing two different clusters. Figure 3.7(b) shows a scenario where the two data items are co-located in cluster c_1. In this scenario, user u_a issues requests r_1 and r_2 at times t_1 and t_2, respectively, resulting in both requests accessing the same single cluster c_1. This represents a scenario with high request locality, fulfilling both its requirements.

The key idea behind DAC is to group data together when it is frequently accessed together. Intuitively two data items that are frequently accessed by a large set of users are related in some manner. This relationship may be syntactic, semantic, relational, structural, geographical, temporal or social. The power of DAC lies in identifying these relationships and their strengths, regardless of the cause. A strong relationship between two data items signifies that they are being frequently accessed together. Therefore in the presence of temporal locality, we can deduce these data items are also more likely to be accessed together in the future.

Example (online social network). Two social network users, *Jane* and *Bob*, retrieve their latest feeds, which contain a status update from two common friends, *Steve* and *Phil*. Not long after, Jane decides to retrieve her latest feed again only to find Steve and Phil's status updates still listed. Therefore in clustering Steve and Phil's two status updates together, we can ensure Jane and Bob's future (repeated) requests for data are handled by a single cluster i.e. all of the data items retrieved by these requests are in one single cluster.

Example (web search engine). A web search engine user u_1 submits the query "london startups" resulting in 10 data items returned $d_1...d_{10}$. Not long after, user u_1 repeats the same query only to receive the same 10 data items again. Therefore in clustering data items $d_1...d_{10}$ together, we can ensure the future (repeated) requests of user u_1 are handled by a single cluster.

Furthermore, these correlations also provide access predictions i.e. users requesting for new data items that have never or rarely been requested for in the past. The intuition is if user u_1 accesses the same data item d_1 as user u_2 does, u_1 is more likely to access other data items user u_2 has accessed over other random data items. This property becomes more dominant with further data items accessed by both users, representing the increased alignment of their interests.

Example (online social network). Our two social network users, *Jane* and *Bob*, retrieve their latest feeds, which now contain a new status update from their common friend, *Steve*. Later Steve posts a further update, with Jane updating her latest feed shortly after. In the past, Bob has been interested in what Jane has accessed and is therefore more likely to access Steve's latest status update than that of a random user. In clustering Steve's two status updates and any further generated ones together, we can ensure Jane and Bob's future requests for data are handled by a single cluster.

Example (web search engine). Two web search engine users, u_1 and u_2, submit the query

$^{User}/_{Data}$	u_a	u_b	u_c	u_d	u_e	u_f	u_g
d_1 (DFV)	3	8	5	2	7	0	10
d_2 (DFV)	5	10	0	6	2	6	2
$d_1 + d_2$ (AFV)	4	9	2.5	4	4.5	3	6

Table 3.1: Example of two data frequency vectors (DFV) and their average frequency vector (AFV)

"best incubators in london" resulting in the same 10 data items $d_1...d_{10}$ returned to both users. Later user u_1 submits the query "best london start-ups" resulting in data items $d_{11}...d_{20}$ returned. In the past user u_2 has been interested in the same data items as user u_1 and therefore is more likely to submit a query resulting in data items $d_{11}...d_{20}$ being returned. In clustering data items $d_1...d_{20}$ together, we can ensure the future requests of user u_2 are likely to be handled by a single cluster.

A similar approach to DAC can be found in machine learning for recommendation engines called collaborative filtering (CF), which was introduced by the Tapestry system [GNOT92]. Recommendation engines typically analyse large click-stream and user rating datasets in an *offline* process, providing users with curated suggestions (or recommendations) for unseen items. A *user-based* collaborative (UB-CF) technique is used in the GroupLens system [RIS94], which considers other similar users for recommendations. This differs from *item-based* collaborative filtering (IB-CF) techniques, where similar items are found according to user preferences [SKKR01, Kar01]. Example applications of recommendation engines in industry are Amazon's [LSY03] use for product recommendations, Netflix's [Kor09] use for film recommendations, and Last.fm's[14] use for music recommendations.

Although our DAC approach resembles some of the mechanisms found in IB-CF, it differs in that it is performed (1) online, (2) on the request dataset (rather than user defined ratings), (3) across a distributed dataset and (4) with the goal of improving latency. These differences will become more apparent in Chapters 4 and 5, where we put DAC into practice for our two application scenarios.

3.3.1 User request frequencies

To implement DAC, user request frequencies must be recorded for each data item, gathering a snapshot of the workload at a point in time. Each data item is therefore associated with a *data frequency vector* (DFV), an n-dimensional vector that contains the frequencies of requests received for a given user at each dimension. Table 3.1 shows two example DFVs for data items d_1 and d_2 and users u_a-u_g. The table shows that data item d_1 was requested seven times by user u_e and data item d_2 only twice by u_e. The table also demonstrates how a collection of data items can be represented using an *average frequency vector* (AFV), in this case consisting of d_1 and d_2. An AFV is calculated as the sum of values at each

[14] Last.fm, `http://www.last.fm/`, last accessed: 20/6/2013

vector dimension, divided by the number of DFVs. AFVs provide us with a summarised representation of all the data items in a cluster, which we use to perform cluster-cluster comparisons and data-cluster comparisons (see Sections 4.3 and 5.2.2).

A potential limitation of the DAC approach is the number of dimensions that are required to store user request frequencies in DFVs and AFVs. As we have demonstrated, each dimension represents an individual user that has performed a request. Vectors can therefore grow to be large over time, resulting in excessive overhead in recording requests. Dimensionality reduction is a well studied problem and various solutions have been proposed [Fod02, CP97]. Techniques include the Karhunen–Loève transforms [Kar47, Loi48] (also know as principal components analysis (PCA) [Hot33] and Latent Semantic Indexing (LSI) [DDL+90]), which are typically used in information retrieval systems, collapse as many dimensions as possible while reducing the amount of information loss.

We opt for a simpler technique to solve this problem. We limit the number of requests recorded, thus minimising the maximum possible dimensionality. We discuss this approach further in Section 3.3.3. Another possible solution to reduce the vector dimensionality in our DAC approach is to cluster **users** according to their data access patterns. The clustering of users is orthogonal to that of data items and provides a reduced representation for the number of similar users in a single dimension.

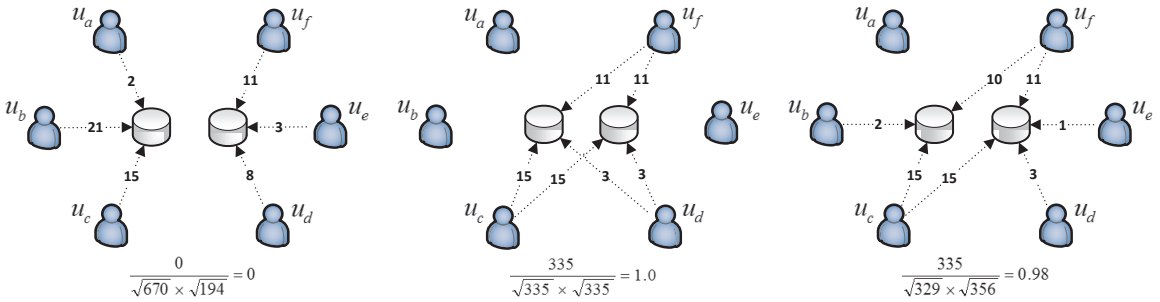
3.3.2 Cosine similarity metric

To be able to cluster data according to access request patterns, we must compare DFVs using a distance or similarity metric. There are a large number of metrics formalised with distinct properties [DD06]. Of these metrics we choose the *cosine similarity* (or Orchini similarity [Och57], angular similarity, normalised dot product) due to its simplicity, high efficiency when operating on sparse data and better results with high-dimensional data. It has been shown that cosine similarity performs better than other metrics such as Euclidean, Pearsons correlation or extended Jaccard on high dimensional data [SGM00]. Furthermore, Markins et al. show the cosine similarity outperforming other measures when used for IB-CF purposes.

Lastly, the cosine measure is a simple calculation, simplifying the similarity calculation process. For example, further studies have shown that IB-CF also yield better quality results and performance with an *adjusted cosine* measure [SKKR01]. Although an *adjusted cosine similarity* has said to provide the best results, it requires the entire client state to be calculated. This is infeasible in a distributed environment where client state is partitioned throughout.

$$cosine\_similarity(i,j) = cos(\vec{i},\vec{j}) = \frac{\vec{i}\cdot\vec{j}}{\parallel\vec{i}\parallel\times\parallel\vec{j}\parallel} \tag{3.4}$$

Used in positive space, cosine similarity measures the cosine angle between two vectors $\vec{i}$ and $\vec{j}$



(a) Zero Correlation (b) Perfect Correlation (c) Near-perfect Correlation

Figure 3.8: Cosine similarities between two data items in three distinct scenarios

yielding a value of 1 when the cosine is 0 ° (parallel) and 0 when the cosine is 90 ° (perpendicular). The measure accounts for the *orientation* of vectors and not magnitude because vectors are normalised during calculation. Equation 3.4 defines how the measure is calculated for two vectors $\vec{i}$ and $\vec{j}$ with unit vectors in the denominator. Note that '$\cdot$' denotes the dot-product of the two vectors.

The cosine similarity can be used to compute the similarity in access patterns between two DFVs, identifying the presence of a relationship. Figure 3.8 shows how the cosine similarity measure is applied to request frequencies when two data items, d_1 and d_2, are accessed by six users, u_a-u_f, in three distinct scenarios. In Figure 3.8(a), the two data items are accessed by two disjoint sets of users, with zero overlap between the data frequency vectors for d_1 and d_2. As a result the cosine similarity measure calculates a correlation of zero (i.e. the two vectors are perpendicular to one another). In this scenario, there does not exist a single user that has accessed both data items d_1 and d_2, and thus they are unlikely to be accessed together in the future. Figure 3.8(b) shows the two data items being accessed by identical sets of users, with perfect overlap between the data frequency vectors for d_1 and d_2 and a correlation of one. In this scenario, all users who were interested in data item d_1 were equally interested in data item d_2, and vice-versa. Finally Figure 3.8(c) shows the two data items with a near-perfect correlation of 0.98, with only a few requests separating the two DFVs. In this scenario, most of the users who accessed data item d_1 also accessed data item d_2, and vice-versa.

As illustrated, the intuition behind DAC is to provide a measure of similarity between data items based on the access patterns of users. This allows us to assess if data items are related without further knowledge of the content, the application or the users. Being able to measure the relationship between data items without knowing the cause is a powerful *general* abstraction, allowing us to classify data for any type of online service or application. Furthermore, in using the frequency of requests from a user's perspective—found in the dimensions of DFVs and AFVs—the similarity measure can provide data clusters with high request locality. We describe this in further detail and with experimental results in Section 3.4.

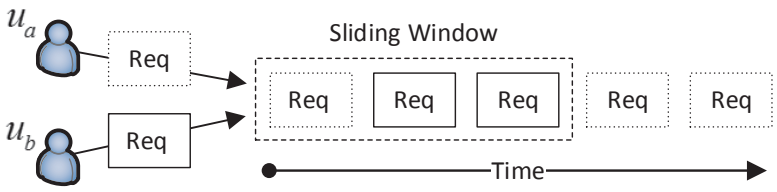


Figure 3.9: Updating of a DFV with a sliding window ($window_{DFV} = 3$)

3.3.3 Workload adaptability

The benefits of statically clustering data items according to historical access patterns can only be achieved for a static workload. In a static setting, previous access patterns leading to a given clustering configuration remain relevant. Under a dynamic workload, however, the cluster configuration must *adapt* to the request workload.

For this to occur, we must update the data frequency vectors. We achieve this with a *sliding window* that ensures that only the last k number of user requests are recorded within each DFV ($window_{DFV} = k$). This achieves adaptability and sets a maximum bound on the number of dimensions in a DFV, and thus the overhead incurred. Although data items are likely to be accessed by a small subset of users, DFVs will become denser as time progresses due to the accumulation of request frequencies. This is a beneficial property of DAC as it has been reported that IB-CF techniques do not perform well with sparse datasets [LSL12]. However, a maximum bound on the number of recorded frequencies is still required to ensure that DFVs can adapt to the current workload and that memory consumption is reduced.

Figure 3.9 shows how a sliding window operates in a DFV, illustrating two users, u_a and u_b, that access a single data item d_1 intermittently. The window size is set to three requests, showing how older requests are discarded due to more recent incoming requests. Requests that are within the window are included in the DFV of the data item as a frequency for that particular user; those that are outside are excluded.

The chosen window size provides a trade-off between the stability and adaptability of correlations within clusters. The larger the window, the more requests recorded in the DFV, making it less prone to the effects of outliers and variations over shorter periods of time (stability). However, DFVs with a large window are slower to adapt to changes in the workload (adaptability).

Determining the appropriate sliding window size k for each scenario requires preliminary testing and evaluation. Scenarios with high throughput workloads require a larger window size to ensure stability, whereas scenarios with low or bursty throughput workloads require smaller window sizes to ensure the correlations adapt within a timely manner. The *dynamicity* of workloads is another dimension which affects what window size to adopt. For example, if there are a large variation of requests in a workload, the correlations between data items will consistently be changing regardless of the workload's throughput. It is because of dynamicity and throughput that various sliding window sizes should be evaluated

before deployment. While the window size parameter can be seen as a limitation of the Data Activity Correlation approach, as we have discussed in Section 5.6, a fixed k parameter would make the DAC approach less versatile.

By recording all user requests made to data items within the sized window, DAC can provide correlations based on the full history of the workload up until the window size. Depending upon the application scenario, this may not be necessary. Instead, DAC can be augmented with a sampling mechanism that randomly sheds requests with a given probability. Sampling may be ideal for handling high throughput workloads or popular data items.

The sliding window technique is also a solution to the limitation we described in Section 3.3.1. A sliding window reduces the number of requests that are recorded and thus minimises the maximum dimensionality of DFVs and AFVs. With this, the worst case scenario for DFVs involves k distinct users performing a single request to a data item, which gives a vector of k dimensions (where $window_{DFV} = k$). AFVs for clusters of n DFVs can have a much larger dimensionality. This presents itself if, for example, each data item performed their worst cases and there was zero user request overlap between data items. Such a case would lead to a worst case dimensionality of kn for a given AFV. However in practice AFV dimensionality is small because DAC clusters data items with similarly overlapping DFVs, thus reducing the dimensionality of the combined DFVs and hence the AFV.

3.4 Experimental exploration

Next we provide an exploration of DAC, demonstrating its ability to identify correlated data and provide request locality through clusters of correlated data. We achieve this by investigating DAC under synthetically generated and real-world datasets. The section is divided into two parts: online application workloads (Section 3.4.1) where we explore DAC under two real world traces—Facebook social interactions and Last.fm track listening; followed by synthetic request workloads (Section 3.4.2), illustrating how user-data interactions are used to construct clusters consisting of improved request locality.

Our experimental methodology involves simulating synthetic and real-world workloads consisting of multiple interactions between users and data items. From these interactions, the DFVs of data items are filled with user request frequency counts and compared with other DFVs. We use the cosine similarity measurements between data items to construct a weighted undirected graph, defined as a *data activity correlation graph* or simply DAC graph. The graph consists of nodes, representing data items, and edges, representing the correlations between data items. Our analysis of these DAC graphs shows how the DAC approach can identify correlated data items and provide high request locality.

3.4.1 Online application workloads

To asses the effectiveness of DAC applied to real-world workloads handled by online applications and services, we investigate with two datasets: Last.fm, an online music discovery service and Facebook. These datasets give an indication as to the conditions that we might expect in our application scenarios. The Last.fm application has an *open* restriction model (restriction models in applications are discussed in Section 3.1.3), allowing users to access (or, in this case, listen) to any track in the entire stored collection. Although the factors for interaction may differ, this models our global sensor discovery platform where users have access to all the sensor data sources in the network. Facebook, on the other hand, has a *restricted* application model, with users only given access to application data that relates to themselves and their direct social connections. Users are therefore more likely to interact with fewer data items, thus making these interactions more predictable.

Facebook social interactions

Researchers have paid much attention to popular services such as Facebook and Twitter due to their global adoption and size. However there has been a general lack of available data for analysis due to legal implications.[15,16] Many researchers have therefore had to gather their own datasets using scrapping techniques [WBS+09, KLPM10, MMG+07].

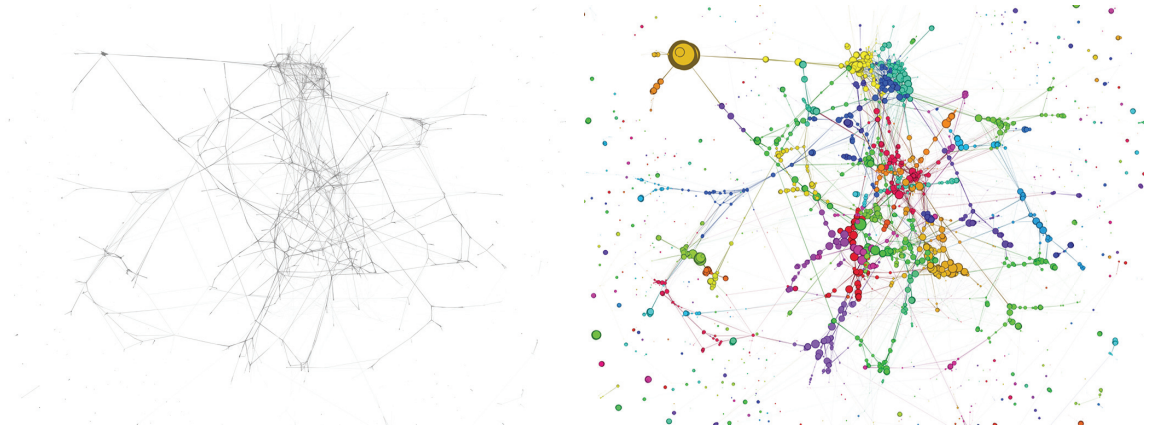
For the purpose of our DAC exploration, we use the dataset obtained by Wilson et al. [WBS+09, WSPZ12]. In this dataset, various social and interaction graphs are gathered by analysing network memberships. Users can belong to various networks, such as schools, institutions and geographical regions, and in 2008 membership was unauthenticated and open to all users. This has since changed, assigning even greater value to this dataset. In crawling these networks, Wilson et al. gathered both social and interaction graphs over different time-spans. While social graphs provide the social ties between users according to friendship lists, interaction graphs provide the subsets of friends that users interact with in a given time-frame.

Although the items in the datasets are not timestamped, the data is divided into three separate parts: first month, first six months and first year. In each of these parts, the order of the items is in chronological order. We therefore perform our analysis of DAC using the interaction graph dataset that spans the first single month (beginning in April 2008).

In order to translate the social network interactions into application data requests, we assume that each node i in the graph is both a user u_i and data item d_i. Intuitively a social interaction with another user requires the fetching of their application data. Therefore when an interaction occurs between users u_a and u_b denoted as $interaction(u_a, u_b)$, we record a request issued by user u_a to data item d_b denoted as $request(u_a, d_b)$. As the

[15] Twitter API ToS, `https://dev.twitter.com/terms/api-terms`, last accessed: 21/5/2013

[16] Facebook Automated Data Collection ToS, `https://www.facebook.com/apps/site_scraping_tos_terms.php`, last accessed: 21/5/2013



(a) Vanilla visualisation of DAC Graph

(b) Visualisation of DAC Graph with communities detected as colour and weighted degree as node size

Figure 3.10: Visualisation of DAC graphs from the Facebook interaction dataset

graph is undirected, we also record the reverse request $request(u_b, d_a)$, generating two user requests for every interaction in the dataset. This has the rationale that a social interaction requires both parties to retrieve the application data of one another eventually. The one month dataset contains a total of $1,412,252$ interactions (and therefore $2,824,504$ generated requests) between $9,268$ unique users to $9,268$ data items.

With user request frequencies recorded in DFVs, a weighted undirected graph can be generated consisting of data items as *nodes* and the calculated cosine similarities between the DFVs of data items as *weighted edges*. The lack of an edge between nodes indicates a similarity correlation of zero between data items. This DAC graph can then be analysed to determine if user request access patterns have a clustering or community tendency.

To detect communities within DAC graphs, we use the *modularity* measure [New06] (denoted as Q), which compares the fraction of edges within a community with the *expected* fraction of edges in a reference graph, also known as a *null-model*. The reference graph is a random graph, which maintains the same degree distribution as the original graph. The resulting measure has an interval of $[0, 1)$, with high positive values indicating a strong community structure. Modularity can be applied to both undirected non-weighted and weighted graphs—the latter being the relevant case for this application. The algorithm used to calculate modularity throughout this thesis is by Blondel et al. [BGLL08] and is an approximation algorithm proposed to handle the analysis of large graphs and networks.

Visualisations can often be helpful in analysing and understanding larger complex networks and graphs that would otherwise be hard to describe. Gephi[17] [BHJ09] is an open-source visualisation framework with a variety of plug-ins for graph layout, analysis and partitioning. Figure 3.10 shows two visualisations of the DAC graph for the Facebook dataset, generated using Gephi and presented using the ForceAtlas2 graph layout algorithm [JHVB11]. ForceAtlas2 is a fast layout approximation algorithm for large graphs, which can include the

[17]Gephi Open-source Visualisation Toolkit, `https://gephi.org/`, last accessed: 22/5/2013

Metric	Value	Metric	Value
Node count (N)	9,268	Modularity (Q)	0.97
Edge count (E)	49,790	Communities detected	640
Average degree	10.7	Connected components	576
Average weighted degree	8.5	Clustering coefficient	0.924
Graph density	0.001	Weighted cluster coefficient	0.871

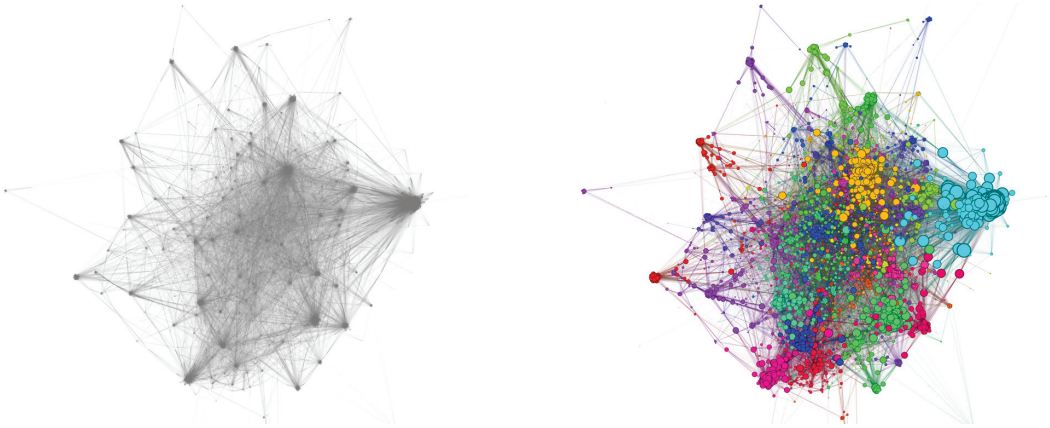
Table 3.2: DAC graph properties for Facebook interaction dataset

weightings on edges to depict distances between nodes.

Figure 3.10(a) presents a vanilla visualisation of the DAC graph showing the natural clustering of data items without any colour-coded partitioning. The nodes in the figure represent data items and the edges between nodes represent their cosine similarity. The shorter the edges between nodes the higher the similarity between them, with the absence of an edge representing a similarity of zero. Figure 3.10(b) illustrates the same DAC graph, highlighting the communities found using the modularity algorithm with different colours, and the weighted degree of data items with the size of nodes. From the figures, we can see clear communities of data items, which have a high similarity in their user access patterns. The larger nodes within the graph show a tight clustering with multiple other similar nodes. Both figures show a large number of disconnected sub-graphs, also known as *connected components*, indicating that Facebook users only interact with a small subset of data items (or other users in this case). This is also confirmed in the larger sub-graph, consisting of multiple larger clusters of nodes interconnected through only a small number of edges.

Table 3.2 summarises the measured properties of this DAC graph, showing a modularity of 0.97 with 640 communities. This indicates a strong community structure, backed up with a cluster coefficient of 0.924 and a weighted cluster coefficient [BBPSV04] of 0.871. Furthermore many of the communities detected are disconnected from one another with a connected component measurement of 576, indicating that interactions on Facebook are highly clustered and localised. Further evidence for this is the fact that the average degree is only 10.7, which indicates that data items are only similar to a small number of other data items, offering opportunities for them to be partitioned. Few edges between data items are of a high similarity, with an 8.5 average weighted degree.[18] Lastly, due to the high level of restriction imposed by the application, depicting which data items can be accessed by which users, the graph is sparse with a graph density of only 0.001. Graph density measures the proportion of edges within the graph compared to a complete graph, i.e. a graph in which every pair of distinct nodes is connected by a unique edge.

[18] As weights are between 0.0 and 1.0, the maximum value that the average weighted degree can be is the average degree i.e. all weights are 1.0.



(a) Vanilla visualisation of DAC Graph

(b) Visualisation of DAC Graph with communities detected as colour and weighted degree as node size

Figure 3.11: Visualisation of DAC graphs from the Last.fm track listening dataset

Last.fm track listening

Last.fm is an online music discovery service with 30 million active users as of March 2009. It recommends tracks according to listening habits and tastes through a recommender system, Audioscrobbler.[19] A dataset of its users' listening habits was released known as the Million Song Dataset [BMEWL11] followed shortly by the Million Song Dataset Challenge [MBMEL12], encouraging researchers to design an algorithm that can improve the outlined recommendation metrics. Two versions of this dataset were released: the more commonly cited "360K users" data set with the number of plays agglomerated into a single item with no temporal information; and a dataset consisting of "1K users"[20] with their complete play history and temporal data included. For the purposes of evaluating DAC, we require the decoupling of requests as well as temporal ordering and therefore have chosen to work with the 1K dataset throughout.

The full 1K Last.fm dataset consists of $19,150,868$ items with fields (1) *user*, (2) *artist*, (3) *song* and (4) *timestamp*, and represents the whole listening habits of 992 users [Cel10]. Data collection was achieved using the Last.fm API and the provided $user.getRecentTracks()$ method.[21] We take a single month sample of this dataset from January to February 2008, which consists of under half a million entries and includes approximately $10,000$ tracks accessed.

Figure 3.11 shows two visualisations of the DAC graph generated using Gephi, which, as for the Facebook dataset, are also presented using the ForceAtlas2 graph layout algorithm. The nodes in the figure represent data items and the edges between nodes represent their cosine similarity. The shorter the edges between nodes the higher the similarity between them, with the absence of an edge representing a similarity of zero. Figure 3.11(a) presents a vanilla

[19] Audioscrobbler, `http://www.audioscrobbler.net/`, last accessed: 19/5/2013

[20] Last.fm Dataset—1K users, `http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-1K.html`, last accessed: 21/5/2013

[21] Last.fm API, `http://www.last.fm/api/show/user.getRecentTracks`, last accessed: 22/5/2013

Metric	Value	Metric	Value
Node count (N)	10,000	Modularity (Q)	0.811
Edge count (E)	468,474	Communities detected	110
Average degree	93.7	Connected components	64
Average weighted degree	55.5	Clustering coefficient	0.894
Graph density	0.009	Weighted cluster coefficient	0.885

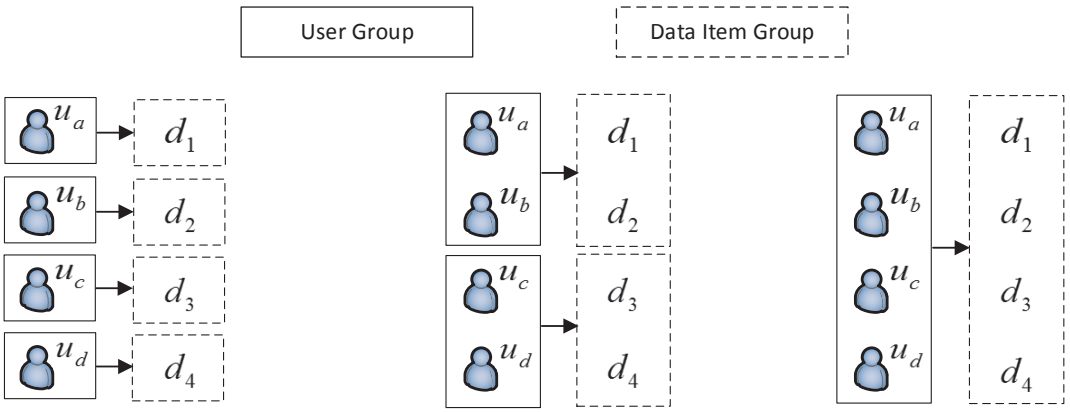
Table 3.3: DAC graph properties for Last.fm dataset

visualisation of the DAC graph showing the natural clustering of tracks without any colour-coded community detection. Figure 3.11(b) shows the same graph with the modularity communities coded as colours and the node sizes representing their weighted degree. We can see that this dataset generates a denser DAC graph compared to that for the Facebook dataset in Figure 3.10. This is due to the open restriction model of the application with users able to listen to any track stored in the system. Irrespective of this, the figures show a clear clustering of tracks with high similarity in their user request frequencies. Figure 3.11(b) shows this prominently with coloured partitions consisting of large nodes. Large nodes indicate strong clusters as the weighted degrees between them are high.

Table 3.3 lists the metrics that were measured for the Last.fm DAC graph. As mentioned, the density of this graph is higher than that of the Facebook DAC graph, with a measure of 0.009. This density is also apparent in the average degree of 93.7, which indicates the data items are related to many other data items. However the similarity measures between these data items are moderate, as evidenced by the average weighted degree of 55.5. The modularity is calculated as 0.811, and 110 communities are found, which indicates a moderately strong community structure in the DAC graph. Of these 110 communities, 64 are connected components, again fewer than with the previous dataset. However both the modularity and clustering coefficient (0.894) indicate that, although the graph is dense with many weaker relationships between data items, there is a strong clustering of these data items. This can be seen in the visualisation of the DAC graph in Figure 3.11.

3.4.2 Synthetic request workloads

Next we explore how various synthetic user-data interaction workloads affect the structure of DAC graphs. This investigation validates our DAC approach and ensures that under known conditions the correct and expected clusters of data items are formed. Furthermore, we examine how these clusters affect request locality, a key property that we wish to exploit in data placement and discovery strategies. Request locality involves the reduction in the number of clusters accessed by each individual user over time.



(a) Zero overlap $r = 0.0$ (b) Half overlap $r = 0.5$ (c) Full overlap $r = 1.0$

Figure 3.12: Request workload overlap parameter for clustering experiment

Clustering

To verify that the correct clusters are formed under known conditions, we use synthetically generated workloads. These workloads encompass a range of user request patterns according to a given parameter. The experimentation is performed by simulating D data items $d_1 ... d_D$ being accessed by U users $u_1 ... u_U$.

The *request workload overlap* parameter, denoted r, has a range interval of $[0.0, 1.0]$. With a workload parameter of $r = 0.0$, each user belongs to a single unique *user group* accessing a *data item group*, each initially consisting of a single distinct data item. In this scenario there is no overlap in the requests of users. With each increment of the workload parameter, we merge all user groups and their corresponding data item groups with another arbitrary user group and its corresponding data item group. This results in half the number of user and data item groups, all of which are equally sized. With each of these iterations, users within each group issue requests to *all* the data items in their corresponding mapped data item group. With a workload parameter of $r = 1.0$, all users have been merged into one single user group, accessing all data items within a single data item group. In this scenario, there is full overlap in the requests of users, with all users accessing all the data items.

This process is illustrated in Figure 3.12 with $U = 4$ users and $D = 4$ data items. Figure 3.12(a) shows the request workload overlap parameter set to zero ($r = 0.0$), with each user placed in a distinct user group and each data item within a distinct data item group. Each user group performs a request to all the data items within a single unique data item group. In this scenario there are no overlapping requests i.e. when users request for the same data item. Figure 3.12(b) shows the request workload overlap parameter set to half ($r = 0.5$), with each user group and data item group from the previous figure merged with another of its kind. Each user in a user group performs requests to all the data items within a single unique data item group. In this scenario there are half the number of groups resulting in a number of overlapping requests. Lastly, Figure 3.12(c) shows the request workload overlap

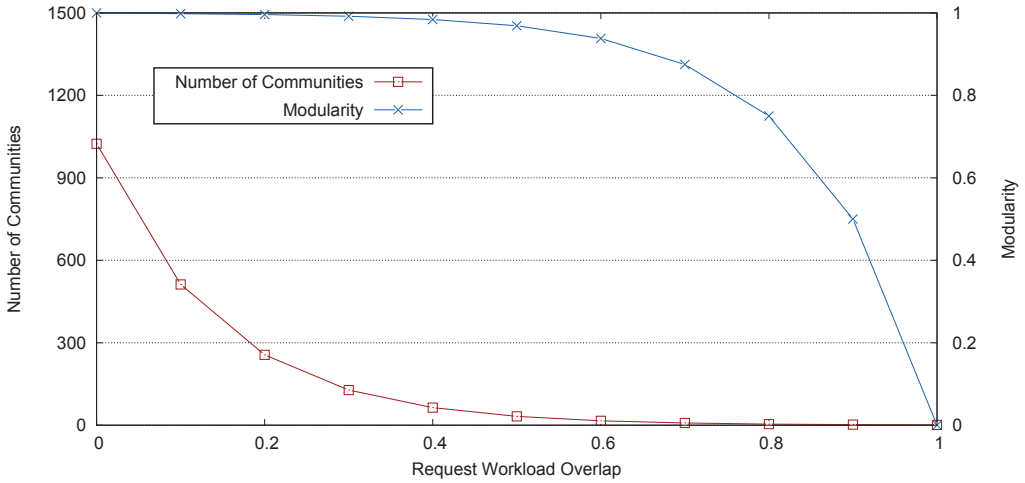


Figure 3.13: Modularity and number of detected communities found with increasing request workload overlapping

parameter set to full ($r = 1.0$), with all the user groups and data item groups merged. Each user in the user group performs requests to all the data items within the only data item group. In this scenario all the requests performed are overlapping requests.

The overall effect of the request workload parameter increasing, is therefore to halve repeatedly the number of user and data item groups by merging them with one another. The experiment begins with no overlap in user requests and increments the number of users that have similar data access behaviour until all D users are interested in all U data items uniformly. After each iteration, the DAC graph is constructed and analysed for comparison.

We perform our experimentation with $U = 1024$ users and $D = 1024$ data items. Figure 3.13 shows the modularity metric on the right y-axis and the number of detected communities on the left y-axis against the request workload overlap parameter. Modularity and the number of detected communities is calculated on the DAC graphs generated according to the workload parameter. The figure shows that the number of detected communities decreases as we increase the parameter. More precisely, the number of communities halves with each iteration—beginning with 1024 separate communities with zero overlap and reaching a single large community under the complete workload overlap of 1.0.

Similarly the modularity decreases with increasing workload overlap—as the difference between the number of edges in communities to that of the null model tends toward zero. When reaching zero, there is a single large community holding all the edges within the DAC graph. Although not shown, the number of *connected components* in the graphs also follows the number of communities found. This indicates that the communities detected are perfectly formed and disconnected from other communities.

This investigation shows the expected behaviour in the formation of data item clusters. In varying the number of data items accessed by users, we observe different sized communities within the DAC graph. More importantly, we show the perfect formation of these clusters without a single edge *between* them. This analysis supports our argument for the effectiveness

of our DAC approach, and its ability to cluster data items appropriately.

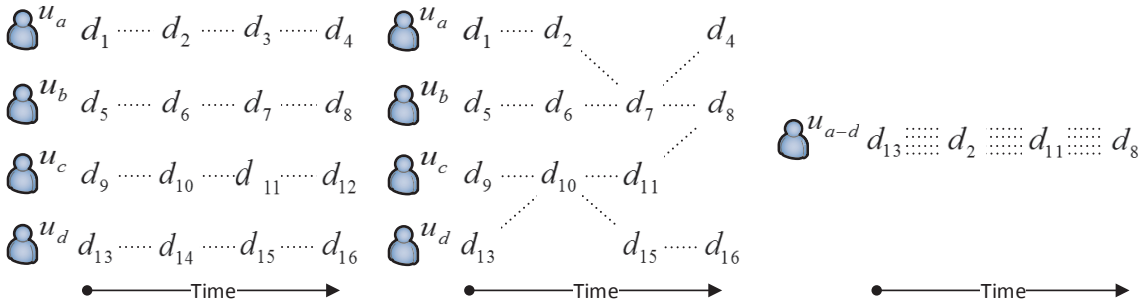
Request locality

A key property required in the clusters formed by our DAC approach is *request locality*. Request locality is a more restricted form of partition isolation, which is beneficial to a variety of areas such as data partitioning (see Section 2.3.1), distributed caching [ABCD96] and memory management [DS72, Smi82] (where it is commonly known as spatial locality). Our motivation for providing high request locality is to reduce the latency experienced by users when accessing stored data in a multi-site wide-area networked infrastructure. We discuss how request locality can be applied to both data placement and discovery strategies in Section 3.5.

The DAC approach performs predictions and exploits *temporal locality* in the workload to achieve request locality. It does this by adjusting the correlations between data items according to past access patterns. Temporal locality refers to the repeated behaviour of users requesting the same data items over time. The intuition is that in the presence of temporal locality, clusters of correlated data items, formed according to past access patterns, can provide request locality for future user requests. As interactions between users and data *evolve* and are not random transformations, there is an overlap between one evolutionary step and another. This overlap is what defines the temporal locality of a workload.

In order to assess the quality of our DAC approach, we evaluate its ability to form clusters with high request locality under various synthetic workloads. Our evaluation is performed using a workload generator, designed to synthesise a wide range of access patterns for a given set of users and their requested data items. It achieves this by progressively increasing the amount of user request overlap in the workload at each round, defined as an *overlap stage*. The generator operates by iteratively collapsing the separate requests of users into multiple identical requests. The generator begins with zero overlap, where each user requests multiple, yet distinct data items, over a time period or *session*. At each overlap stage or overlap increment, the workload generator removes one of these data items being requested and replaces it with a data item requested by another user, i.e. increasing the number of overlapping data items being requested. This process is repeated until full overlap is reached, where all users are requesting the same data items. We define the entire process of generating workloads with zero overlap to full overlap as a single *run*. The generator differs to the one presented previously in that this overlap is at a finer granularity and is non-deterministic i.e. overlaps are randomly chosen and thus arbitrary.

The intuition behind the workload generator can be seen in Figure 3.14 where we illustrate the overlap process for four users u_a–u_d, initially requesting four distinct data items each (d_1–d_4, d_5–d_8, d_9–d_{12}, d_{13}–d_{16}). Each line in the figure represents the data items that must be retrieved for a particular user in that round over time. The retrieval of multiple data items may be due to multi-get requests (see Section 2.1), the result of multiple single requests issued over time, or a combination of them. In either case, the effect on the stored frequencies



(a) Zero overlap (b) Three overlaps (c) Full overlap

Figure 3.14: Synthetic overlap workload for access locality experiment

in DFVs is the same. Figure 3.14(a) shows the initial setup for the workload generator with zero overlap, resulting in all four users requesting for four distinct data items. Figure 3.14(b) shows the configuration after the workload generator has performed three random overlaps in the requests of users. Data items d_3, d_{12} and d_{14} have been replaced by data items requested by other users. Figure 3.14(c) shows the configuration of the workload generator with users requesting the same four data items, i.e. full overlap.

The goal of the DAC approach is to generate k partitions or clusters of data items with a configuration that maximises the request locality. Request locality decreases the number of clusters that are requested by each user in a session, defined as *multi-cluster* requests. For our evaluation, we assume each cluster consists of p data items ($p = n/k$), where n is the total number of data items. At each overlap round of the workload generator, the requests are conceptually turned into DAC graphs, which are then clustered using the popular *k-means* algorithm [Mac67, For65] discussed in Section 5.2.1. The standard k-means algorithm generates k of clusters of arbitrary size, rather than clusters of fixed size p. We therefore use a variation of the algorithm for the purpose of this evaluation which provides fixed sized clusters.[22]

In order to gauge the quality of DAC clusters, we compare these with clusters generated by two other strategies: *random* and *optimal* at each stage in a run. For the random strategy, we place all data items randomly across the k clusters. However to finding the optimal cluster configuration to this problem—i.e. a division of data items—is NP-complete [AR04]. We implement a brute-force approach using dynamic programming, which limits our evaluation to a small number of data items and clusters. More specifically, we evaluate DAC request locality using the same set-up depicted in Figure 3.14. This consists of four users requesting four distinct data items each, and thus a system with a total of sixteen data items ($n = 16, k = 4, p = 4$).

$$\frac{\prod_{i=1}^{k-1} \binom{n-ip}{p}}{k!} \tag{3.5}$$

[22] Equals K-Means Clustering in Java, `https://code.google.com/p/ekmeans/`, last accessed: 21/5/2013

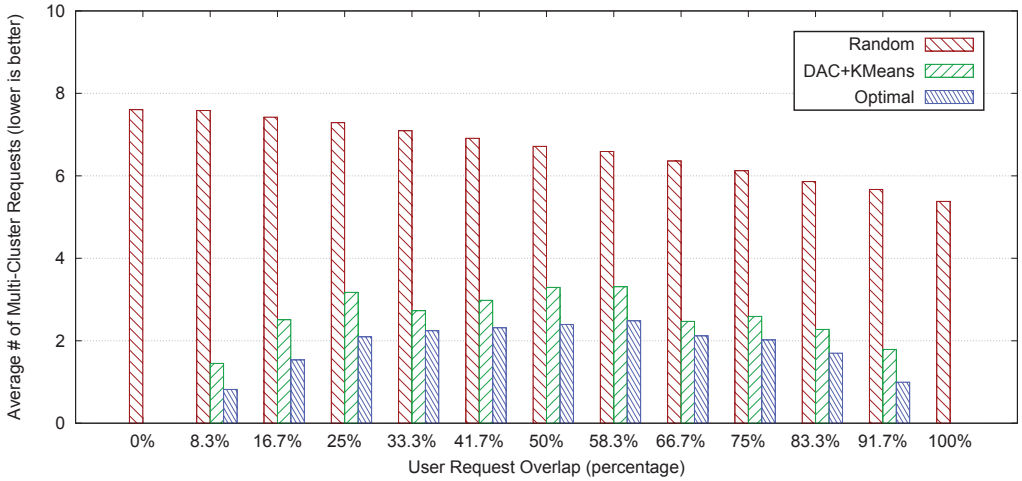


Figure 3.15: Comparison of clustering strategies against the average number of multi-cluster requests

Equation 3.5 specifies the total number of combinations that exist for a given scenario. The equation takes the product of p-combinations from a set of $n - ip$, fixing p data items into a cluster with each increment of i, until $k - 1$ iterations are complete. p-combinations in a set of $n - ip$ is equal to the binomial coefficient $\binom{n-ip}{p}$. The intuition behind this is to fix repeatedly p data items until we have one cluster left with known data items i.e. the data items that remain. We divide this product by $k!$ to eliminate the different orderings of clusters which have identical cluster configurations.

For the parameter values chosen in this experiment, there are a total of $2,627,625$ combinations i.e. unique cluster configurations. The comparison with the optimal strategy requires the computation of *all* configurations at each overlap stage to search for the optimal configuration. This is a time consuming operation. To speed-up this process, we pre-compute and store all these configurations. When determining the optimal configuration for a given workload, we search through the stored configurations evaluating their request locality and returning the highest performing.

In the experiment we conduct 200 runs. Each run involves multiple overlap stages, starting from zero to full overlap in the request workload, generated by the workload generator. At each overlap stage in a run, the request workload is evaluated. Evaluation is performed in three steps: (1) the generation of DAC graphs from the DFVs and the k-means clustering of data items based on these graphs; (2) the search for an optimal configuration that minimises the number of clusters accessed by each user; and (3) the random configuration of data items across clusters for the comparison to a random configuration strategy. The results reported for each overlap stage are averaged over the 200 runs performed.

Figures 3.15 shows the number of multi-cluster requests performed by users with increasing overlap. The number of clusters accessed is large for the random configuration of data items, slowly declining with increasing overlap. This decline is due to the increasing probability of co-locating all data items requested on the same cluster. The optimal configuration

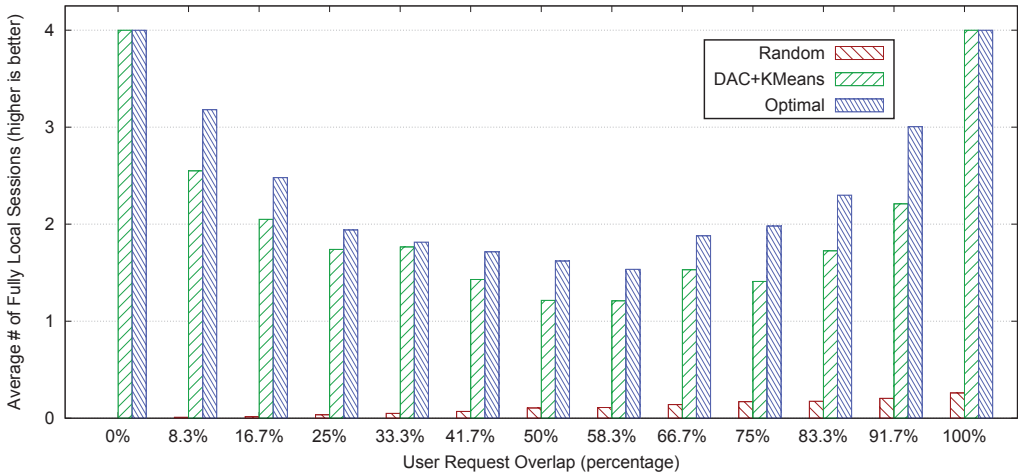


Figure 3.16: Comparison of clustering strategies against the average number of user request sessions with perfect request locality

shows a negative parabola curve, beginning and ending with zero multi-cluster requests and peaking between 50% and 58.3%, with an average of 2.5 requests. Zero multi-cluster requests indicate that all users obtained their data items from a single cluster throughout their sessions. The reason for the shape of the curve is the nature of overlap generation—as users begin requesting distinct data items, the optimal solution is to place these items into their own clusters. Similarly, at the highest overlap, all four users request the same four data items, and the optimal solution places these four items in a single cluster.

The results for our DAC approach together with the k-means clustering algorithm shows that it can generate cluster configurations that are comparable to the optimal solution. For the extreme values of the overlap parameter, DAC+K-means finds the optimal solution leading to an average of zero multi-cluster requests. Therefore, with a given clustering algorithm, the DAC approach provides a good approximation to the optimal solution for reduced multi-cluster requests.

Figure 3.16 shows the number of user request sessions that have perfect request locality i.e. where all the requests issued by a user are handled by a single cluster. While the number of multi-cluster requests shown in the previous figure is related, it does not demonstrate how many user request sessions have perfect request locality. For the extreme values of the overlap parameter, we can see that the optimal configurations result in perfect request locality. This reduces with a trough at 50% of the overlap parameter, where the workload generator produces heavily overlapping access patterns. Our DAC+K-means clustering strategy follows the same pattern as that of optimal. Finally the random strategy shows that only a few runs provide perfect request locality. These runs occur when the user request overlap is at its highest, thus reducing the total number of item that are requested and increasing the chances of them being clustered together.

The experimental exploration of the DAC approach demonstrates its ability to identify relationships between data items according to the access patterns exhibited. When coupled

with a clustering algorithm, the clusters formed provide high request locality, which can be exploited to provide low latency access to data in a distributed setting.

3.5 Application to data placement and discovery

The DAC approach provides a heuristic for clustering data that have similar access patterns. We have demonstrated that clusters generated with DAC provide a number of properties. In this section we discuss how DAC clusters and their properties can aid dynamic data placement and discovery strategies operating across a multi-site networked infrastructure.

Data placement. The objective of a placement strategy is to place data across sites in order to shorten the network path lengths between users and the requested data. The problem can therefore be seen intuitively as a clustering one. In clustering data and placing clusters onto sites that are closest to users, an increased number of requests can be handled locally. This results in fewer requests issued to other globally distributed sites in the infrastructure, and thus lower network latencies. The appropriate clustering of data onto sites can be achieved with our DAC mechanism, which optimises for request locality. We explore how our DAC approach can be applied to data placement in Chapter 5.

Data discovery. The objective of a discovery strategy is to forward user requests across sites (each storing statically placed data) in a manner which shortens the total network path length between users and the requested data. The informed forwarding of requests across a network depends on the application-level links between sites, which define the structure of the overlay network. An intuitive solution to the data discovery problem is to form logical networked clusters of sites. Users can then be assigned to their most appropriate cluster, reducing the number of network hops that occur during data discovery. Reduced hop count, shortens the network path length, thus improving user response times. The clustering of sites—and the data that they store—can also be achieved with our DAC approach, reducing the number hops traversed during discovery. We explore how our DAC approach can be applied to data discovery in Chapter 4.

A key component to our DAC approach, and thus the data placement and discovery strategies applying DAC, is its ability to adapt to the current workload. Workload awareness is achieved by leveraging the similarity in access patterns between data items. Our experimental exploration has thus far used the standard k-means algorithm for clustering data items, an offline clustering algorithm. However for clusters to adapt with the current workload, the DAC approach must be coupled with an *online* clustering algorithm. We leave the discussion of online clustering for Chapters 4 and 5, where we present our solutions for two application scenarios.

3.6 Summary

We began this chapter by analysing the workloads of existing online services and applications, identifying four major factors that influence user-data interactions. These include properties of (1) the users, which exhibit both diverse and dynamic characteristics; (2) the applications, which have rapidly evolving software and application logic; (3) the interactions, which involve semantic, geo-spatial and temporal locality and their frequent shifts; and (4) the data, which is dynamic, shared amongst multiple users and interdependent on other data. This analysis demonstrated the complex, global and dynamic nature of data access patterns.

The next section argued for the need of a workload-aware mechanism that is both generic and adaptable, and thus capable of handling the properties of workloads we have outlined in the previous section. We proposed clustering as a suitable approach to performing data discovery and placement in a multi-site wide-area networked infrastructure. The data clustering mechanism proposed has a number of requirements: request locality, application-agnostic, adaptability and access prediction.

We introduced data activity correlation (DAC), an approach for correlating data items according to the similarity of their access patterns. DAC achieves this with two data structures, data frequency vectors (DFVs)—storing the request frequencies for an individual data item—and average frequency vectors (AFVs)—storing the request frequencies for a group or cluster of data items. Correlations between data items are then calculated as the cosine similarity between DFVs. We showed the adaptability of DAC, evolving DFVs with the current workload and thus the correlations between data items.

We performed an experimental exploration of the DAC approach, under both synthetic and real-world workloads. We then demonstrated its ability to produce well-formed clusters of data items with high access correlation in open and restricted applications. Furthermore we showed the ability of DAC to identify clusters of near-optimal request locality for a variety of access patterns.

The chapter concluded with a discussion of the DAC approach and its application to data placement and discovery in Section 3.5.

Chapter 4

Data Discovery: Self-Organising Global Sensor Space

For the advancement and realisation of the 'Internet of Things' (IoT), we investigate the development of a global sensor discovery platform. Such a platform must deliver fresh results to users with low response times. A major contributor to user response time in such a global environment is the network latency involved in routing discovery requests, as discussed in the problem statement in Section 2.1. For a system to reduce this delay, it must adopt an appropriate data placement and discovery strategy. In this chapter, we focus on the development of a **data discovery strategy** for multi-site wide-area networked infrastructures that can provide users with fresh, low latency results.

We realise our data discovery strategy through the Global Sensor Space (GSS), a global sensor discovery platform that connects a sea of sensors and data sources into a single unified and self-organising infrastructure. GSS organises data sources into a hierarchical cluster-based overlay network, with clusters of data sources formed using our DAC approach from Chapter 3. An online and distributed *split-and-merge* clustering algorithm keeps data sources organised into DAC clusters according to the current request workload, providing high request locality. This ensures that GSS is able to provide fresh and accurate results to users with low response times.

We begin the chapter with Section 4.1 which explains how the DAC approach can be used to develop a workload-aware data discovery strategy. The section then shows how this applies to the global sensor discovery problem and introduces Global Sensor Space. Section 4.2 provides further detail on the design and architecture of GSS. In Section 4.3 we present the distributed clustering algorithm used in GSS to keep DAC clusters organised. We then present an evaluation of the methods used in GSS in Section 4.4 followed by a discussion in Section 4.5.

4.1 Data discovery with DAC

A global sensor discovery platform must interconnect the data sources of the world into a unifying infrastructure, such that the requests of globally distributed users are handled accurately—using fresh metadata to query against—and with low response time. This is essential in providing a good user experience and ensuring that the autonomous applications and actuators built on top of such a platform can react in a timely manner (see Section 1.1.2). We realise this infrastructure with the construction of an *overlay network* [LCP05]— a logical network of nodes and links built on top of an underlying physical network, which in this case is the Internet. Before describing the intuition behind the platform's overlay network structure, we outline our assumptions and requirements.

4.1.1 Assumptions and requirements

Each data source or node within the infrastructure is an Internet-connected PC that either manages: (1) a single connected sensor; or (2) multiple sensors and data sources aggregated at a single local base station node, such as commonly found in wireless network deployments [MMR+08,MMG+08,LGBS09]. These data sources are therefore *sinks* for sensors that are deployed in proximity and are responsible for maintaining their metadata. Unfortunately there are multiple standards for describing sensors and the data they produce [BPRD08], presenting us with metadata heterogeneity. For example, metadata for one temperature sensor can be constructed with different attribute names (such as "temp" or "temperature" and "Celsius" or "Fahrenheit") and standards (such as the Sensor Model Language (SensorML) [BR07] and the Transducer Markup Language (TransducerML) [Hav07]) compared to that of another temperature sensor.

```
<gml:description>Water cooler sensors in office</gml:description>
<gml:identifier codeSpace="uid">imperial:huxley:room346:54987</gml:identifier>

<sml:outputs>
  <sml:OutputList>
    <sml:output name="watercooler">
      <sml:DataInterface>
        <sml:data>
          <swe:DataStream>

            <swe:elementType name="watercooler_data">
              <swe:DataRecord>

                <swe:field name="time">
```

```
            <swe:Time definition="http://sensorml.com/ont/swe/property/
                SamplingTime">
              <swe:label>Sample Time</swe:label>
              <swe:uom xlink:href="http://www.opengis.net/def/uom/ISO
                  -8601/0/Gregorian" />
            </swe:Time>
          </swe:field>
          <swe:field name="temp">
            <swe:Quantity definition="http://sensorml.com/ont/swe/property/
                AmbientTemperature">
              <swe:label>Water Temperature</swe:label>
              <swe:uom code="Cel" />
            </swe:Quantity>
          </swe:field>
          <swe:field name="level">
            <swe:Quantity definition="Waterlevel">
              <swe:label>Waterlevel</swe:label>
              <swe:uom code="cm" />
            </swe:Quantity>
          </swe:field>

        </swe:DataRecord>
      </swe:elementType>

      <swe:encoding>
        <swe:TextEncoding tokenSeparator="," blockSeparator="&#32;"
            decimalSeparator="." />
      </swe:encoding>

      <swe:values>
        2013-06-02T10:00:25Z,18.3,20.6
        2013-06-02T10:00:35Z,18.4,20.6
        2013-06-02T10:00:45Z,18.4,20.0
        2013-06-02T10:00:55Z,18.4,19.8
        2013-06-02T10:01:05Z,18.3,19.8
      </swe:values>

    </swe:DataStream>
  </sml:data>
 </sml:DataInterface>
</sml:output>
```

```
  </sml:OutputList>
</sml:outputs>

<sml:position>
  <gml:Point gml:id="stationLocation" srsName="http://www.opengis.net/def/crs/EPSG
      /0/4326">
    <gml:coordinates>51.5 0.18</gml:coordinates>
  </gml:Point>
</sml:position>
```

Listing 4.1: Example data source metadata in SensorML 2.0

```
<swe:values xlink:href="http://sensors.imperial.ac.uk:4563/room/346" />
```

Listing 4.2: Alternative sme:value tag for SensorML 2.0 metadata

The platform requires syntactic conformity for the data source metadata, with sensor data in the platform expressed using a consistent standard. Although not restricted to any one, SensorML [BR07] aligns with our design decisions and is able to capture the measurements of a wide range of sensors both static and dynamic. Furthermore these measurements are expressed using the Extensible Markup Language (XML), an easily understood, versatile and human-readable format. XML-based metadata can be queried using well-defined query languages such as XQuery [BC03].

Listing 4.1 shows the SensorML 2.0 metadata for an example GSS data source, which consists of two sensors in a water cooler: a water temperature sensor measuring in Celsius, and a water level sensor measured in centimetres. The listing shows the last five measured values stored in the metadata, allowing for fast processing of conditional queries. For example, a query to discover all temperature sensors that have a reading below 21 °C. An alternative method to embedding sensor readings into SensorML 2.0 metadata is to provide a link to, for example, a RESTful (Representational State Transfer) [Fie00] web service. Listing 4.2 shows an alternative *swe:value* tag to Listing 4.1, consisting of a link to the water cooler data source.

While the platform requires the syntax of metadata to be homogeneous, metadata semantics can be varied. Although coordination of attribute names within sensor metadata improves the user experience of the platform, it limits the description of data sources by forcing agreement on a predefined namespace and dimension. We therefore do not enforce this as a requirement but rather recommend that the attribute names of new data sources overlap with those already existing if they have the same meaning. This design choice provides greater flexibility to the user and differs from existing solutions. For example, semantic overlay networks (SON) [CGM05, DV10] (see Section 2.5.5) organise data sources according to the various *types* of sensors, such as temperature, pressure and humidity, from a predefined set. Listing 4.1 shows an example GSS data source with the metadata for both a predefined

sensor type (Temperature) and a custom sensor type (Waterlevel).

Our final assumption comes from the requirement of freshness. Each sensor or data source provides a raw data stream which is defined as a (potentially) unbounded sequence of data items, also known as tuples. These are either ordered explicitly, with a timestamp, or by the values of one or more data elements (for example a packet sequence identifier in an IP session). All data streams are a continuous and sequential stream of data items, each with its own set of characteristics: (1) input rate, ranging from a few bytes to a few gigabits per second; (2) irregular and bursty data; and (3) varying types of data elements, structured in a variety of ways.

A high data rate means sensor metadata is continuously being updated with the latest observations, making it impractical to decouple the metadata from the data source. Replication or caching of metadata require consistency to be maintained through repeated cache replacements and update requests. Any delay in performing these replacements/updates would lead to user requests being executed against out-dated metadata and potentially returning inaccurate results (see Section 2.2.1).

Furthermore data sources may only be willing to provide data to trusted or paying users. Thus, by caching or replicating, data sources are no longer in full control of their data. For example, caching the latest measurements from a radiation sensor in a nuclear plant has security implications. Although caching and replication is critical for popular data, we focus on the low latency discovery of the actual original data.

4.1.2 Overlay networks

With data sources managing their own sensor metadata, any reduction in network latency depends on the data discovery strategy. Data discovery in a multi-site wide-area networked infrastructure involves the organisation of sites into an overlay network, followed by the appropriate forwarding of requests over this structure. An efficient and low latency strategy optimises the number of times a request is forwarded over the overlay network (or hops), reducing the final network path length and forwarding time. This differs from a data placement strategy where data can be decoupled from its site and therefore migrated or placed onto another site closer to user accessors.

There are various existing overlay network designs and algorithms for performing search over these structures, as outlined in Section 2.5. One possible design is a *network-aware overlay network* [NZ02] where the organisation of nodes and links between them reflects the underlying physical network properties. The advantages of such a structure is its low or near-zero stretch and therefore efficient routing of messages [PLMS06]. Figure 4.1 shows how a network-aware overlay network constructed using, for example, network coordinates, applies to our application scenario. The figure shows how the global distribution of data sources in the physical network is mirrored in the overlay network shown above.[1]

[1] Although done in practice, we do not wrap the overlay network around the globe for illustration purposes.

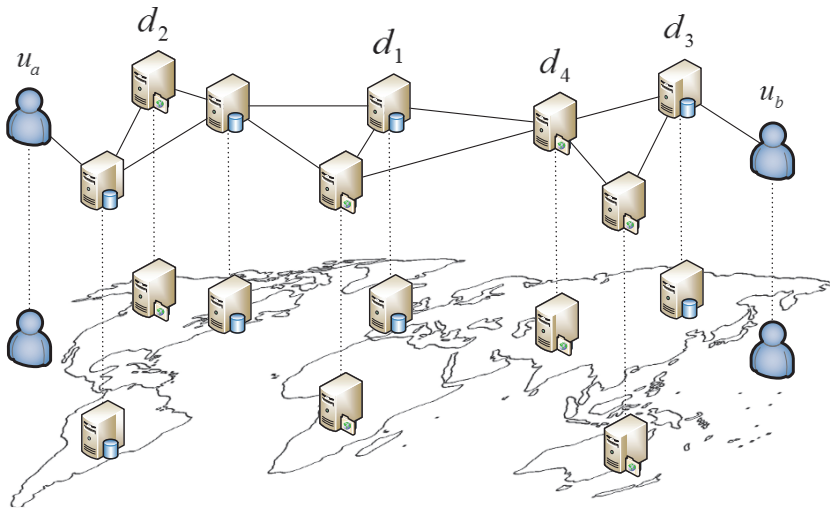


Figure 4.1: Global data sources organised into a network-aware overlay network

Although the forwarding of messages between data sources in this overlay network is fast and efficient, it can result in multiple hops and data sources visited. For example if user u_a issues a request to the system, which results in the discovery of data source d_3 and d_4, at least four links will be traversed and five data sources visited (assuming no caching or replication of metadata as discussed) using the breadth first search (BFS) algorithm (see Section 2.5.3). The same applies with user u_b issuing a request resulting in the discovery of data sources d_1 and d_2. Although the traversal of each individual link provides low latency, the combined path length and additional application-level forwarding of requests results in higher overall response times for users. Taking our previous example, if user u_a had a direct link to data source d_3 and d_4 in the overlay network, the combined path length is likely to be shorter and thus provide user u_a with a lower response time.

An alternative to such an organisation of nodes and links is to adopt a *semantic overlay network* (SON) [CGM05] where data sources are either classified, clustered or discovered according to the similarity of their content or metadata (see Section 2.5.5). Figure 4.2 shows an example of a SON with the same globally distributed data sources arranged into groups according to their sensor type metadata. The same scenario, involving users u_a and u_b issuing requests for the discovery of d_3+d_4 and d_1+d_2, respectively, also results in multiple data sources and links being traversed, and thus leads to high user response times. This overlay network has high stretch, where *stretch* [AMD04] defined as the ratio between latency on the overlay network and latency on the physical network. A lower stretch therefore provides lower response times and reduces unnecessary consumption of network resources.

Semantic overlay networks have high stretch regardless of the geo-spatial locality in the workload. For example, although user u_a is geographically close to d_3—shown in Figure 4.1—and would expect to have low latency access to this data source, requests must traverse multiple data sources and links before reaching d_3.

Other structured overlay networks such as DHTs [SMK01] (see Section 2.5.4) rely on the construction of a distributed index based on key-value pairs. This is not a suitable solution

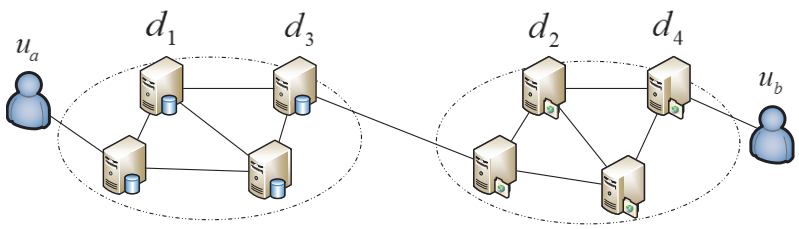


Figure 4.2: Global data sources organised into a semantic overlay network

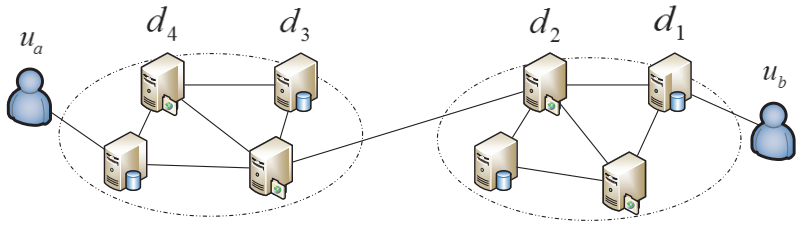


Figure 4.3: Global data sources organised into a workload-aware overlay network

for our application scenario due to limited query model provided to users. Users must submit requests with *exact* matching keys to the data sources that they wish to discover. Although discovery can be achieved within $O(logn)$ number of hops, these are of high network latency due to the random distribution of values across data sources.

The organisation of links and data sources in the overlay network is therefore essential to achieving low latency. While network-aware overlay networks have low stretch links, users' frequently requested data sources may be located far away (i.e. low geo-spatial locality workload). Similarly, while the organisation of nodes into a semantic-based structure can assist with the discovery of data sources (with respect to an attribute in their metadata), requests may have an increased hop count, each with high network latency due to high stretch.

4.1.3 Global Sensor Space

A more intuitive method for constructing an overlay network, which we propose, is to use the current request workload. A method for achieving this is to cluster data sources that are frequently requested together. This ensures that the number of hops between them are kept to a minimum and that the processing of requests local. Figure 4.3 illustrates an example *workload-aware overlay network* (WAON) with data sources organised according to past requests. Users u_a and u_b submitting their requests to d_3+d_4 and d_1+d_2, respectively, are now able to discovery their respective data sources with fewer overlay hops and therefore a reduced network path length. Such an approach allows for the configuration of data sources according to a variety of workloads, whether requests have semantic locality, geo-spatial locality or even social locality (see Section 3.1). This is because a WAON can adapt its structure according to patterns in the workload, and actively reduce the number of hops performed for requests. Given a workload with high geo-spatial locality, a workload-aware

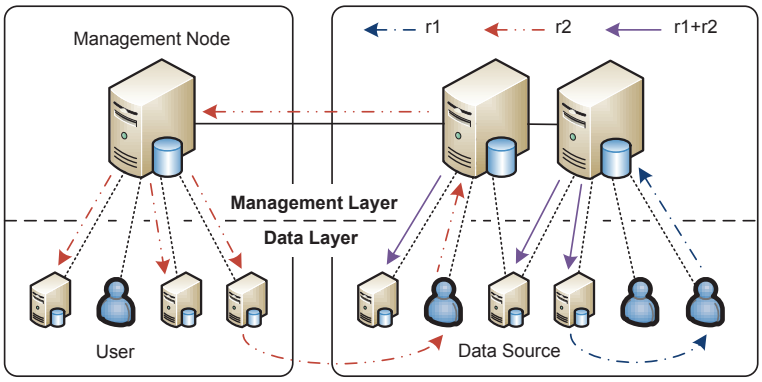


Figure 4.4: System model illustrating the layers of two clusters

strategy will tend toward a low stretch overlay network as the organisation of nodes will closely resemble the underlying physical network.

We use this idea to present Global Sensor Space (GSS), a global sensor discovery platform that interconnects data sources into a workload-aware self-organising overlay network. The organisation of this network is determined according to the *data activity correlation* (DAC) approach from Chapter 3 and is coupled with a distributed online clustering algorithm to ensure its continuous adaptability. This results in clusters of data sources with highly correlated access patterns, which increases the number of requests that can be handled within a cluster (intra-cluster requests) and reduces the number of requests issued across clusters (inter-cluster requests). The system is therefore able to provide globally distributed users with a reduced hop count and low response time when performing data source discovery.

4.2 System design

The GSS infrastructure is composed of a hierarchical clustered overlay network, consisting of regular *data sources*, *management data sources* and *users*. We refer to all three of these entities as generic nodes within an overlay network. These nodes are organised into clusters according to our DAC approach. The intuition behind this structure is to cluster data source nodes that have been frequently requested in the past. A larger fraction of future requests can then be handled by a single cluster and thus reduce the number of overlay network hops. We discuss how the DAC approach is applied to GSS in Section 4.3.

Each cluster has two layers or hierarchies: (1) a *management layer* that consists of a minimum of two management nodes and (2) a *data layer* that consists of the rest of the data source nodes in the cluster. Management nodes are cluster-heads, or supernodes, and maintain links to all regular data sources in their cluster i.e. a star-like topology. This ensures that users which are assigned to a cluster can perform intra-cluster data source discovery with a low number of overlay hops.

Figure 4.4 illustrates the organisation of nodes in GSS and their division into management and data layers. User and data source nodes are placed into *crisp* clusters—i.e. non-

$^{Cluster}/_{User}$	c_1	c_2	c_4	c_6	c_7
u_a (DFV)	7	3	1	11	8

Table 4.1: Example of a Result Frequency Vector (DFV)

overlapping clusters—together, shown as cluster 1 and 2 in this example. Each cluster is then divided into the two layers, the management and data layer. The data layer contains leaf nodes, which include all of the users and the majority of data sources throughout the system. The remaining data source nodes make up the management layer. Any data source node can be promoted to or demoted from management status.

To alleviate potential bottlenecks with single cluster-heads, clusters are kept to a manageable size and have at least two management nodes. Multiple management nodes within clusters remove single points-of-failure improving the fault-tolerance of the system. Clusters with a failed management node pro-actively promote another data source to management status. Any data source in a cluster can be promoted to management node status, although a more sophisticated selection criteria could be used such as with a QoS-based heuristic.

To interconnect clusters, management nodes maintain links in an unstructured topology. The flexibility of this topology provides GSS with a configurable trade-off. The more connected these nodes are, the more efficient routing in the overlay networks becomes. However the downside to increased connectivity is that the additional links must be maintained, and thus has an added overhead.

4.2.1 Data source discovery

The idea behind GSS's discovery strategy is to begin the search locally, from within a user's own cluster, before expanding the search horizon to include other clusters. This ensures that clusters are not overwhelmed with requests that are unlikely to be satisfied by them. Users perform their cluster selection process using the success rates of their past requests, determining which clusters their results were predominantly returned from. We define the data structure used to provide this functionality as a result frequency vector (RFV), structured similarly to that of data frequency vectors (DFV) described in Section 3.3.1.

A DFV stores the number of user requests handled by a data item in a vector, with each dimension representing a user. In a RFV each dimension represents a cluster, and the frequencies stored are the number of results returned by that cluster. Table 4.1 illustrates an example result frequency vector (RFV), showing user u_a receiving results from clusters c_1, c_2, c_4, c_6 and c_7 in the past. The example shows cluster c_6 having successfully handled 11 requests issued by user u_a.

The clustering approach discussed in the following section, maximises the likelihood of a user's request being satisfied within their cluster. The discovery process is illustrated in the previous figure, Figure 4.4, where a user in cluster 2 performs an intra-cluster search (r1

and r1+r2 arrowed lines). A search is performed with the user submitting a request that contains the query that should be run against data source metadata. The request is initially sent to the management node (r1) in the cluster, which then propagates it to all leaf nodes in the local cluster (r1+r2). The relevant data source nodes then contact the user directly to either deliver their metadata or a continuous stream of sensor data.

The organisation of nodes and links is also designed to minimise the number of inter-cluster requests, as they result in additional overlay hops and therefore higher response times for users. Figure 4.4 also shows a user performing an inter-cluster search from within cluster 2 (r2 and r1+r2 arrowed lines). The request is initially forwarded within the cluster (r1+r2), only then to continue to contact others in a broadcast manner. Other management nodes perform a local search of their own clusters (r2). There are limitations to the inter-cluster broadcast approach, which we discuss further in Section 4.5. Again, the relevant data source node contacts the user directly after it has been discovered.

4.2.2 Nodes joining and leaving

Data source nodes join the GSS system by contacting a random *bootstrap node* from a predefined list of available data source and management nodes. These lists are stored in a centralised online service, replicated for fault tolerance and kept separate from the GSS platform. The bootstrap node directs the joining node to either a single (if not a management node) or set of management nodes (if a management node). The node then creates its own singleton cluster and promotes itself to management status.

Once established as a cluster, a *split-and-merge* algorithm, discussed in Section 4.3, can then decide to merge the cluster with another depending upon the current context. However before any merging can take place, the data source must populate its DFV with requests. This ensures that the merging process is not performed prematurely and with another potentially dissimilar cluster. Nodes in GSS may leave the system at any time by will or failure. However, if a management node leaves, the other management node within the cluster promotes another data source to management status.

Users join the GSS platform using the predefined list of available nodes, assigning themselves to a random cluster until they accumulate enough results to perform an informed local cluster choice. A downside of this approach is that users must keep track of request statistics, and thus requires a front-end application to interact with the platform. For example, users must interact with the GSS platform through a website consisting of a Javascript (JS) application. This application can store the user's request frequencies in a result frequency vector (RFV) and repeatedly evaluate which cluster to join.

4.3 Self-organising data source clusters

For a data discovery strategy to be able to provide users with low user response times, the organisation of sites and links must reflect the workload. Deriving a strategy according to an analysis of the workload requires substantial time and effort, and is typically inaccurate. Furthermore, as workloads evolve, discovery strategies may become out-dated, eroding the benefits that they once provided. Therefore GSS has a self-organising overlay network, which depends on the workload itself. More specifically, clusters of data sources continuously adapt their memberships according to the latest similarity in request patterns between them.

Given the *data activity correlation* approach presented in Section 3, we can calculate the relationship between data sources. Two data items that are frequently accessed together by multiple users are likely to be related in at least one dimension, whether this be syntactic, semantic, relational, temporal or geo-spatial. By successfully clustering related data sources in the overlay network, requests are more likely to be satisfied within their origin cluster. The motivation for this is simple: intra-cluster searching is a low latency operation, and fewer nodes need to be contacted to discover a relevant data source, resulting in a lower response time. On the other hand, inter-cluster search requires other clusters to be searched, resulting in both higher response time and increased message complexity.

In this section, we outline the *split-and-merge* distributed clustering algorithm used to group data sources with similar access patterns. Although the notion of a split-and-merge clustering algorithm has been researched in the past [CVKW05], solutions have been centralised, require global knowledge of all the data and restricted to static datasets. More recent work has considered a stream-based split-and-merge technique for evolving datasets [Lug12], but is centralised and again requires global knowledge of all the data. Instead we divide the two mechanisms—split and merge—and delegate these tasks across multiple nodes in a wide-area network.

In GSS, clusters are *crisp*, i.e. non-overlapping, with each node a member of a single cluster at a given point in time. Clusters can either be split or merged, with only the data sources in them affected by these operations. Users must therefore decide which cluster to join independently based on their RFVs. Both operations are execute independently and are performed in a distributed manner: management nodes perform local operations—i.e. within their own cluster—for determining whether their cluster should be *split*; and an elected global management node performs global operations determining whether any two clusters should be *merged*. We discuss each of these two operations separately in the following sections.

4.3.1 Splitting clusters

To split a cluster our algorithm requires an evaluation metric to determine whether a potential cluster division is of better or worse quality. There are two main categories to cluster analysis techniques, internal criterion and external criterion [MRS08]. An external criterion

evaluates a cluster against a benchmark or "gold" standard class, such as a dataset of known correct assignments. Such a criterion therefore requires prior knowledge of the classification of the clustered data items, and therefore is not applicable to our scenario. An internal criterion evaluates the intra- or inter-cluster similarity, assessing the level of *cohesion* (also known as compactness) or *separation*. Such a criterion is able to determine the quality of a cluster, with only knowledge of the data items in a cluster. This is a benefit for our application, as management nodes can asses the quality of their own cluster without requiring global knowledge.

GSS intuitively optimises for increased *cluster cohesion*: it determines how closely related the data sources in the cluster are according to their DFVs. There are a large number of types cohesion measures available [TSK07]. For example, prototype-based clusters calculate cohesion using the centroid of the cluster for computation (such as the sum of squared distances outlined in Equation 5.3, Section 5.2.2); graph-based clusters use the similarity measures between the items within the cluster.

For simplicity, we use a cohesion measure typically found in graph-based clusters for our objective function, though in practice this component of the system is interchangeable. This measure calculates the sum of similarities between all links within a cluster, defined in Equation 4.1. Our particular instantiation of this cohesion uses the cosine similarity in Equation 3.4 for its definition of edge weights, calculated using the data source DFVs stored on the cluster's management node.

$$cohesion(c) = \sum_{\substack{x \in c \\ y \in c \\ x \neq y}} cosinesimilarity(\vec{x}, \vec{y}) \tag{4.1}$$

Split operations are performed by the management node of a cluster. They gather the request frequencies (DFVs) of data sources within their cluster to make informed decisions. However, for a cluster to split, a data source configuration must exist that yields a higher cohesion than the current one. This is possible due to the migration of a single or multiple unrelated data sources to a new cluster. Algorithm 1 shows the process which management nodes execute periodically in rounds.

Management nodes begin the process by constructing a *local* proximity matrix. Each entry in the matrix indicates the similarity between the DFVs of a data source pair and is calculated using the *cosinesimilarity* in Equation 3.4. This process can be seen in lines 1 and 2 of the algorithm. The space complexity of this matrix is $O(n^2)$, however due to symmetry in the matrix it only has a maximum of $|n|\left(|n|-1\right)/2$ entries. While the local proximity matrix is shown to be updated periodically in its entirety (lines 1–4), individual rows and columns are updated as data enters the system. For example, data source d_1 may send its latest data frequency vector (DFV) to the management node at a different time to that of data source d_2.

Next, the average cohesion of the cluster is calculated using the pre-calculated values in the

Algorithm 1: GSS algorithm for splitting DAC clusters

Input: DFVs for data sources within cluster c: $\{\vec{d_1}, ..., \vec{d_{|c|}}\}$, Split threshold: ϕ
Output: Identifying a split for cluster or not and its configuration

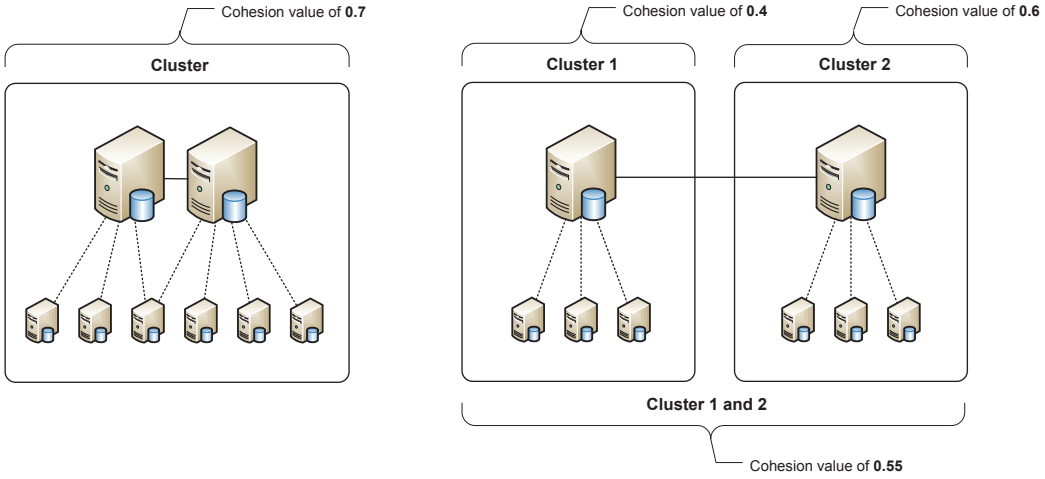
```
1  foreach data source pair (d1, d2) within cluster c do
2      localMatrix[d1][d2] ← cosinesimilarity(d1, d2)
3  coh_c ← cohesion(c, localMatrix)/|c| (|c| − 1) /2
4  maxcoh_c ← coh_c
5  partition_c ← {}
6  foreach partition (c1, c2) of cluster c do
7      coh_1 ← cohesion(c1, localMatrix)
8      coh_2 ← cohesion(c2, localMatrix)
9      avgcoh ← (coh1+coh2) / (|c1|(|c1|−1)/2+|c2|(|c2|−1)/2)
10     if avgcoh > maxcoh_c + φ then
11         maxcoh_c ← avgcoh
12         partition_c ← (c1, c2)
13 if maxcoh_c > coh_c then
14     split(partition_c)
```

local matrix, shown in line 3. This value is then used to initialise $maxcoh_c$ in line 4, along with the initialisation of $partition_c$ in line 5. The algorithm shows the variables for cluster c.

The next phase of the algorithm performs the search for partitions that yield a higher coherence value than that of $maxcoh_c$ plus a given threshold ϕ. The algorithm performs a brute-force search, iterating through all possible split configurations of the cluster, comparing the average sum of both cluster coherences with the last assigned maximum. This approach is clearly not a scalable solution for a large number of data sources, but can easily be replaced with, for example, a clustering algorithm [JMF99,Mac67,For65,DM01,CGC01,Hua98,JD88]. Lines 7–9 calculate the coherences for the given partitions (coh_1 and coh_2). Their average over the sum of edges and links in both partitions is assigned to *avgcoh*. In the same iteration, *avgcoh* is compared with the current maximum coherence value, taking into consideration the provided threshold. Partitions that are of higher coherence are remembered and compared against in the next iteration (lines 11 and 12).

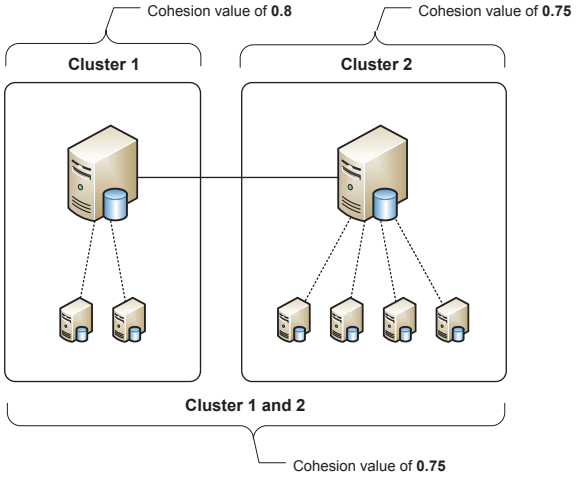
Finally, after having considered all possible partitions, the management node checks whether a new configuration has been found, leading to the execution of the split function (lines 13 and 14).

This split process is illustrated in Figure 4.5 with an example. Figure 4.5(a) shows the original cluster configuration with an average cohesion value of 0.7. From this configuration, various partitions are generated with their cohesion values compared. Figure 4.5(b) illustrates a partition that is not chosen, due to the partition's cohesion of 0.55 being less than that of the current configuration. Conversely Figure 4.5(c) shows a possible partition, with its cohesion value of 0.75, which is higher than that of the current configuration.



(a) Original DAC cluster before splitting

(b) A possible configuration which is not chosen due to a lower cohesion value than the original configuration



(c) A possible configuration which is considered as its cohesion value is higher than the original configuration

Figure 4.5: Example split operation of a DAC cluster using the cohesion measure

4.3.2 Merging clusters

An ideal merge algorithm would mirror the objective function of the split algorithm. In this case, this would involve calculating the cohesion of clusters individually and comparing this to the cohesion when merged. However, with this strategy, the elected management node requires global knowledge of all data source DFVs in the system. Although the objective function alignment for these two algorithms would provide higher cluster cohesion, it would introduce considerable communication overhead.

For this reason, we investigate an alternative approach for the merge process. Rather than requiring the DFVs from all the data sources within each DAC cluster, we use a single approximated value to identify clusters: their average frequency vector (AFV) or centroid. AFVs

Algorithm 2: GSS algorithm for merging DAC clusters

Input: Cluster centroids; $\{\vec{\mu_{c_1}}, ..., \vec{\mu_{c_{|C|}}}\}$
Output: Identified global cluster merges and their configuration

```
1  sortedMergers ← {}
2  foreach cluster pair (c1, c2) within C do
3      globalMatrix[c1][c2] ← cosinesimilarity(μ_c1, μ_c2)
4      sortedMergers ← add(c1, c2, globalMatrix[c1][c2], sortedMergers)
5  coh_C ← cohesion(C, globalMatrix)/|C| (|C| − 1) /2
6  merged ← {}
7  foreach Merge pair (c1, c2) within sortedMergers do
8      if globalMatrix[c1][c2] > coh_C ∧ size(c1, c2) ∧ c1 ∉ merged ∧ c2 ∉ merged then
9          merge(c1, c2)
10         merged ← add(c1, merged)
11         merged ← add(c2, merged)
```

are composed of the average of all DFVs within a given cluster as discussed in Section 3.3.1. This reduces both the storage and communication overhead as elected management nodes only require a single vector for each DAC cluster.

For clusters to merge in this alternative way, the similarity of their centroids must be higher than the calculated global cohesion. Similar to our split implementation, we take this approach because it is simple and intuitive. However an alternative internal criterion with reduced complexity could as easily be adopted. The merge algorithm is shown in Algorithm 2, and it is executed on an arbitrarily selected management node in the system.

The algorithm begins with the computation of a *global* proximity matrix for all cluster pairs in the system. This is to calculate the average global cohesion (coh_C) in order to compare with various merged clusters. A global matrix differs from a local matrix in that the elements being compared are AFVs (or cluster centroids) rather than DFVs. With each computation, the value and configuration is stored in a sorted list. This process can be seen in lines 2–4 of the algorithm. Similarity of two AFVs is calculated with the same metric used throughout, the *cosinesimilarity* (line 3).

The algorithm calculates the average global cohesion (coh_C) in the system for comparison using the global matrix of cosine similarities (line 5). The final phase of the merge algorithm is to merge clusters greedily that yield a higher similarity than that of the global cohesion. Line 8 shows the conditions that must be met before two clusters can be considered to be merged: clusters must not have already been merged previously, and their combined size must be within a defined range. Further optimisation to the merge algorithm is possible (but not shown), such as breaking the greedy merge loop when the first pair that is below the global cohesion is returned ($globalMatrix[c_1][c_2] > coh_C$). This is possible due to the merge pairs (c_1, c_2) being sorted in $sortedMergers$.

After the merge conditions are met, the management node informs the two cluster management nodes from the chosen pair (c_1, c_2), and the merging protocol is initiated between them

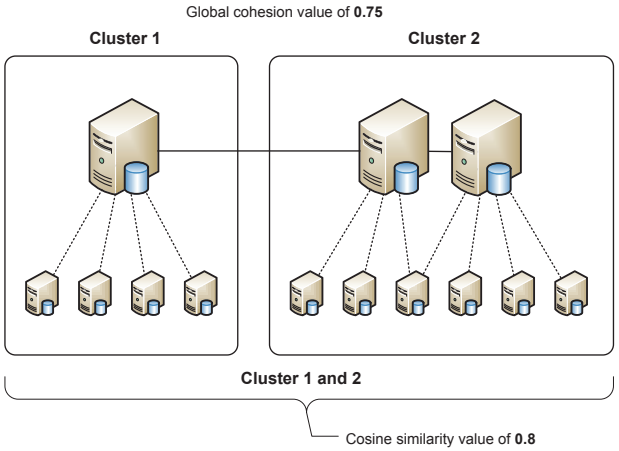


Figure 4.6: Example merge operation of two DAC clusters

(line 9). Cluster management nodes can reject merge operations when they concurrently execute a split operation. As the global management node handling the merge operation only holds summarised information about clusters, it is unable to make an informed decision as to which operation should have preference.

The merge process is illustrated in Figure 4.6, showing an example configuration of two DAC clusters. The figure shows how the merge operation compares the cluster's cosine similarity value (or cohesion based on centroids) with that of the global cohesion value. In this example, the cosine similarity of clusters 1 and 2 is 0.8, which is higher than the calculated global cohesion of 0.75. The algorithm would therefore attempt to merge these two DAC clusters.

Due to the different approach taken by the merge algorithm compared to that of the split, we risk the global configuration of data sources and clusters never converging under a static workload. This is because performing a merge according to the comparison with the average global cohesion always finds a merge pair—there is always a pair that will provide a higher cohesion than the average of all pairs. For convergence, the split algorithm must therefore either find a stable configuration (with a given threshold ϕ) or the merge algorithm must reach the maximum size of clusters. This is a limitation of merging based on average frquency vectors (AFV) of clusters rather than data frequency vectors (DFV) of data item for each cluster. The benefit of this is the reduced communication overhead, due to sending summarised information rather than all cluster DFVs to the elected management node.

4.4 Evaluation

We empirically evaluate the Global Sensor Space system design, demonstrating its ability to construct a workload-aware overlay network and provide users with fresh low latency access to data sources. Our results show GSS can achieve low latency under high geospatial locality and semantic locality workloads compared to a static overlay network. This

section also shows the behaviour of the *split-and-merge* algorithm when faced with a perfectly partitioned workload, correctly identifying clusters of high similarity.

4.4.1 Evaluation methodology

To evaluate the various aspects of GSS and the split-and-merge algorithm, we have implemented a simulator in Java. The simulator is designed to emulate the behaviour of users, data sources and management data sources in the GSS system. Its main objective is to capture the global distribution and typical latencies experienced between data sources and users. To achieve this, we utilise a real-world latency model in order to estimate communication costs over a WAN. This model contains a complete matrix of median latencies between 127 globally distributed nodes.[2]

The dataset was collected using the PlanetLab platform,[3] a global research infrastructure currently comprising of approximately 1166 nodes at 558 sites, which supports the research and development of network services. The platform is used by researchers simultaneously, handling both short-term and continuously running deployments. Its global distribution of sites provides researchers with a way to develop and test large-scale distributed systems, in addition to studying the properties of WANs.

By cross-referencing the latency dataset with the metadata for all PlanetLab nodes, we extract the geographic location (latitude and longitude) of nodes for our system. These locations allow us to pre-compute the geographical distances between all node pairs using the well-known *haversine* formula [Sin84] for future referencing. We use this information to construct high geo-spatial locality workloads, where users are, for example, interested in data sources that are positioned geographically near to them. We discuss this in more detail in the following section.

Our evaluation goals are (i) to investigate the performance of the system under various workloads, and (ii) to validate the split-and-merge algorithm using a pre-defined partitioned workload. All experimental runs of our simulator are performed using the 64-bit 1.7.0_21 Java JVM build on a 64-bit machine with an Intel Core i7 1.73Ghz processor with 8 GB of RAM.

4.4.2 Performance

We evaluate the performance of the GSS design by reporting the user response time. User response time is comprised of multiple factors such as the application-layer processing of data at each node and the delay for entire requests to be sent over a network path (see Section 2.1). Our reported user response time is therefore an estimate based on the summation of network latencies when routing requests between nodes. However, as nodes are globally distributed

[2]PlanetLab Median Dataset, `http://www.eecs.harvard.edu/~syrah/nc/`, last accessed: 10/10/2011
[3]PlanetLab, `http://www.planet-lab.org/`, last accessed: 3/5/2013

and requests/responses in the system are likely to be small, the major contributing factor to user response time is the network latency.

For the evaluation of performance, we divide the set of available nodes into 78 data sources and 49 users, each with its own geographic location. Each experimental run is performed by mapping the nodes to a random simulated PlanetLab node, which is provided with a geographic location and the median latency values to all other nodes. Each user then issues requests to its own group of data sources with a Zipf distribution and 1.0 skew, representing the interests of this user (I_u). We set the number of data sources a user is interested in to 12, which are chosen according to the workload being tested. We chose 12 data sources due to the limited number of nodes we have in the latency model.

The evaluation of GSS is performed under two workloads:

Geo-spatial locality. A high geo-spatial locality workload involves users requesting data that is related to their geographic location. Our analysis of workloads in Section 3.1 described the presence of geo-spatial locality in many current online services and applications such as, YouTube and Yahoo!. These services have an *open* restriction policy to data interaction, with users able to access the majority of the data stored in the system. Furthermore the content stored is typically related to a geographic location.

Our design for GSS has these same properties, allowing users to discover any data source without restriction. Data sources are potentially tied to a geographic location such as an urban pollution detection sensor mounted on a lamp post or a smartphone GPS sensor on the move. It is therefore reasonable to assume that the interactions of users in the GSS system could be influenced by their geographic location.

We emulate geo-spatial locality in a synthetic workload by utilising the geographic locations of nodes and their calculated distances from one another. In this workload, the interest group for each user (I_u) is filled with their closest data sources. During runtime of the simulator, users issue requests to the data sources in this interest group uniformly.

Semantic locality. The workloads of online services and applications also have semantic-based properties (other than geo-spatial locality), such as the interaction with socially or topically-related data. Although these relationships evolve, users have a predefined set of interests that are somehow semantically related to them. In the case of GSS, this may be a user's interest in a *type* of data source or sensor. In order to emulate this semantic locality in our workloads, we populate a user's interest group with a random set of data sources. This is not the same as allowing users to request any data source at random, as we restrict the size of this interest group to just 12 data sources.

Our evaluation compares three strategies: (1) *Static*, where the overlay network is in a random fixed state, without the split-and-merge algorithm executed; (2) GSS, where the overlay network begins with the exact same configuration as the *static* strategy but instead runs the split-and-merge algorithm; and (3) *Direct*, where the user response time of users directly requesting metadata from the correct data source is recorded. A direct strategy

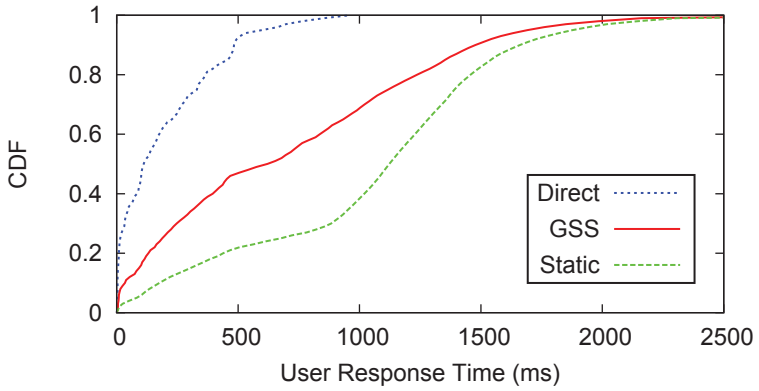


Figure 4.7: Distribution of user response times for three strategies under a geo-spatial locality workload

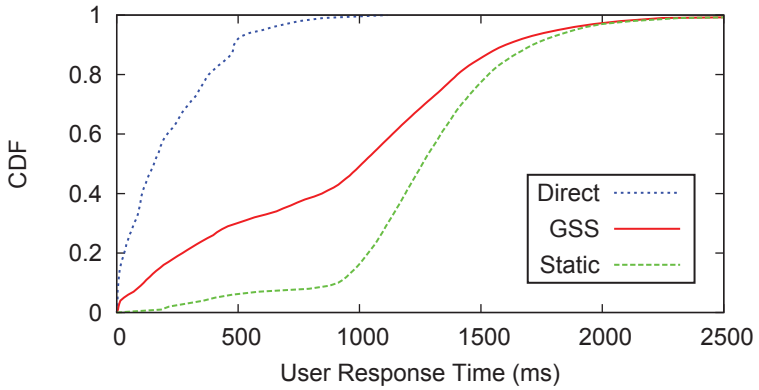


Figure 4.8: Distribution of user response times for three strategies under a semantic locality workload

assumes perfect global knowledge by all users and therefore can by-pass the overlay network entirely. This provides us with a bound on the best possible response time that can be achieved.

The performance results for the three strategies (Direct, GSS and Static) under a high geo-spatial locality workload can be seen in Figure 4.7. The figure shows the average cumulative distribution function (CDF) of user response times for three repeated runs, with 100,000 requests issued by users at a constant rate. In these plots, we see that GSS ($\phi = 0$) provides a median response time speed-up of 45% over a static overlay network structure and 12% for the 85th percentile. As GSS begins with the same overlay network structure to that of the static strategy and does not perform any training beforehand, the higher percentiles remain similar. The figure also shows that with perfect global knowledge, users can satisfy their requests with considerably lower response times, with a median speed-up of 82% over GSS.

Figure 4.8 shows the same experimental set-up as that of the previous figure but under a semantic locality workload. In this scenario, we can see that both GSS and the static approach perform worse than when under a geo-spatial locality workload, with a large fraction of user response times shown to be higher. With the static approach, user requests are forwarded

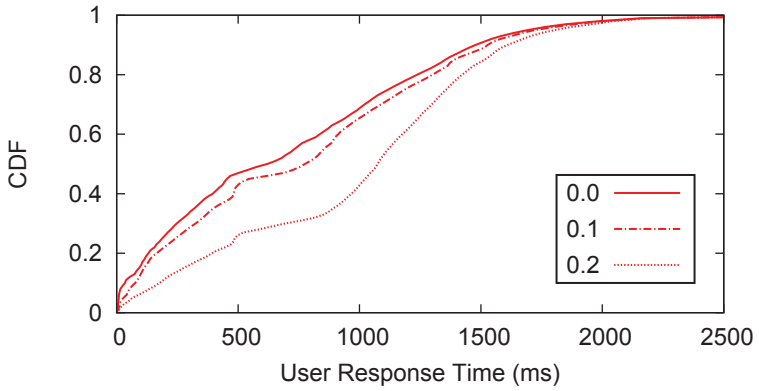


Figure 4.9: Distribution of user response times with varying thresholds (ϕ) under a geo-spatial locality workload

over the overlay network toward data sources that are potentially distant to the user, contributing to the delay. GSS is able to arrange the overlay network according to the semantic relationships in the workload, producing DAC clusters of related data sources. This organisation reduces the number of network overlay hops that are performed during discovery. Because of this, GSS outperforms the static strategy with a 20% improvement in median response times. For lower percentiles, the difference is greater, with a 65% improvement for the 25th percentile.

The organisation of the GSS overlay network in this scenario differs to that under a geo-spatial workload: the stretch of the former is higher. This is because the semantic locality defined in the workload has no relationship to geo-spatial locality, and thus the network latencies between the data sources in DAC clusters are likely to be higher.

The two previous experiments evaluated the various strategies under two diverse workloads with a split threshold (ϕ) of zero for the GSS system. Figure 4.9 shows the cumulative distribution function for user response times with varying thresholds under a geo-spatial locality workload. The data presented is the average of three repeated runs, each executing 100,000 user requests. The figure shows a correlation with the threshold parameter in GSS and the distribution of user response times. When increasing the threshold, GSS performs fewer split operations, which results in fewer merge operations. The system is therefore unable to construct a low latency overlay network structure, with a threshold of 0.2 only just outperforming the static approach.

4.4.3 Correctness

One method for testing a clustering algorithm's correctness is to assess its results using external validation. This is a procedure where the identified clusters are compared using prior knowledge, such as against a known correct configuration. We therefore evaluate the correctness of our clustering algorithm by constructing a *partitioned* synthetic workload. As this evaluation does not require response times, we discard the restriction on the number of

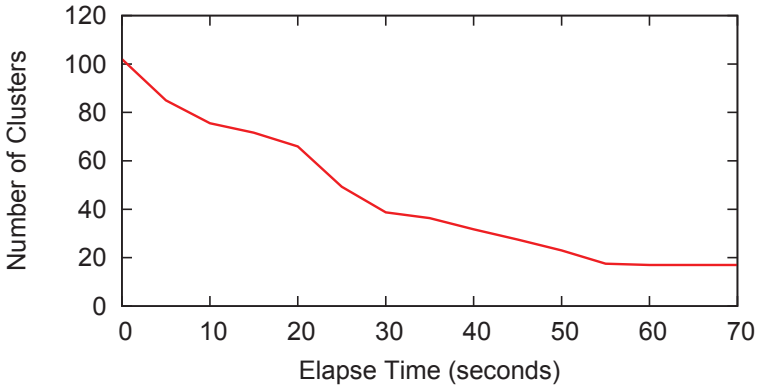


Figure 4.10: Number of clusters in the system against time

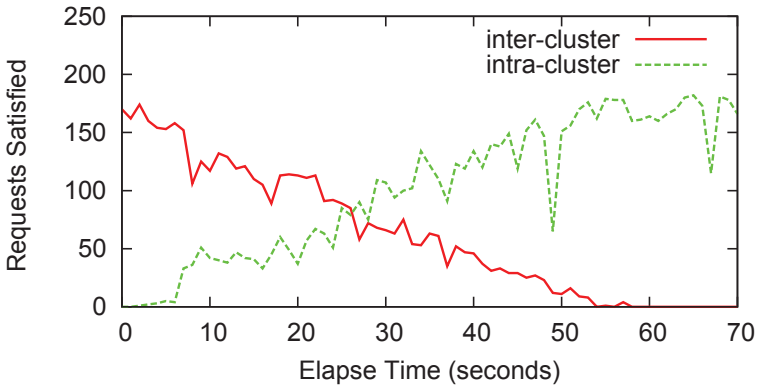


Figure 4.11: Number of inter- and intra-cluster requests satisfied againt time

data sources and users in the system due to the latency model. The experimental set-up has 102 data sources and 1000 users performing a total of 500,000 requests at a constant rate.

The partitioned workload is defined by creating 17 sets of 6 data sources each, with each user randomly assigned to a single (interest) set. User can only request 6 different data sources, all in the same set. This ensures that no DAC similarities are created between data sources in different sets. We expect data sources in the same set to become increasingly related, with their DFV cosine similarities tending toward 1.0 when executing requests. As a result, DAC clusters should be formed around these defined data source sets, which would illustrate the correct behaviour of our clustering algorithm.

Figure 4.10 shows the number of clusters in the system over time. The figure illustrates that GSS correctly identifies the 17 pre-defined clusters, with a steady decrease in the number of clusters. The reason for the algorithm's convergence is due to having a perfectly partitioned workload. Each subsequent operation is able to correctly identify the global optimum configuration.

Figure 4.11 shows the number of user requests that were satisfied from within their originating cluster (intra-cluster requests) and the number of requests that spanned multiple clusters (inter-cluster requests). The figure shows a clear transition in the type of requests

handled, from high hop count inter-cluster requests to low hop count intra-cluster requests. Twenty five seconds into the experiment, Figure 4.10 shows the number of clusters to be approximately halve that of the starting configuration. It is at this point that the number of intra-cluster requests exceeds the number of inter-cluster requests. Once GSS has found the correct partitioned set of 17 clusters, the number of inter-cluster requests remains zero.

4.5 Discussion

The proposed approach presented in this chapter has a number of limitations, which we discuss along with possible solutions.

Scalability. The Global Sensor Space system has a number of components, which could limit scalability. The split-and-merge algorithm distributes split operations across clusters, which enable clusters to act autonomously according to local knowledge. Merge operations are, however, centralised at an elected management node, which issue merge requests to clusters based on summarised global knowledge. This requires the elected management node receiving AFVs from each cluster, which may limit the number of clusters that can be supported by the system.

This could be resolved by distributing the *merge* process using, for example, a gossiping protocol [FPRU90, HKMP96, EGKM04, ACM04, ACMD+03]. Management nodes can perform searches through the management layer in an attempt to find a partner to merge with. This process could be further optimised by loosely enforcing a structure on these links, with management nodes maintaining links to other clusters that they have a high similarity to. A merge operation would then only require a nearest neighbour search to identify if any suitable partners exist.

A further limitation is the way in which *split* configurations are discovered. We describe an algorithm that performs a brute-force combinational exploration of all possible splits. Although these operations are performed separately across the data sources in each cluster, this is not a scalable solution for large clusters of data sources. Other clustering algorithms such as the popular k-means algorithm (see Section 5.2.1) could be integrated as part of the split process, reducing the complexity of searching for clusters considerably.

Request efficiency. Two of the requirements of the system are to provide fresh results and have an expressive query model. Caching, replication or indexing are therefore not suitable for such a system, and instead we force user requests to be executed against metadata stored on the data sources themselves. This inevitably introduces inefficiency as requests must be forwarded to a potentially unbounded number of nodes in the overlay network, due to the broadcast model adopted. This is the classical trade-off between: enforcing a fixed structure in the overlay network, providing efficient request forwarding but limiting the query model to exact matches; and an unstructured overlay network, providing inefficient request forwarding but greater flexibility in the query model.

Our technique for improving the efficiency of requests and keeping a flexible query model, is to stop the subsequent forwarding of requests when they can be successfully satisfied from within a user's local cluster. However, requests that fail to be satisfied from within a user's local cluster must then be broadcast—i.e. as inter-cluster requests—to other clusters. This broadcast approach for inter-cluster communication is inefficient and is open to future work. Providing both a flexible query model and efficient requests could be achieved by replacing the inter-cluster broadcasting model with a more sophisticated approach. Requests could, for example, be forwarded across clusters with a given probability, such as with modified BFS [KGZY02], or broadcast with increasing diameters, such as with iterative deepening [YGM02]. Alternatively, inter-cluster requests could be forwarded in an informed manner, where they are directed toward clusters that are most likely to respond (see Section 2.5.3).

PlanetLab latency model. The latency model used to evaluate GSS was gathered from PlanetLab, which is known to have restrictions [SPBP06]. For example, network links between PlanetLab sites do not necessarily represent those commonly found on the Internet. Many of the sites are part of the global research and education network (GREN), consisting of high-speed access links across the world. The latency model may therefore inaccurately represent ordinary commercial links. Furthermore the geographic distribution of PlanetLab nodes is skewed, with many sites agglomerated in a few regions. GSS provides its latency benefits when nodes are widely dispersed and have large distances between them. We therefore expect GSS to have further performance benefits when under such conditions.

Finally, the latency model must also be a complete matrix to ensure that we can model network latency between all nodes. In computing this complete matrix, the number of nodes was reduced to 127. This limits the scale GSS is evaluated at.

Workload datasets. Another limiting factor is the lack of a real-world workload dataset. A suitable dataset would provide a time-series log of user requests issued to various data items. Ideally these data items would be globally distributed and contain the geographic location of both users and data items. This would allow us to identify the geo-spatial locality properties of the workload. A latency model between all of the entities—i.e. users and data sources—would also be ideal, due to the inaccuracies obtained when mapping the user-data interactions to another latency model. For example, a user in the workload located in Los Angeles, CA would need to be mapped to the geographically closest node in the latency model. Depending on the similarity of the workload and latency model, this may be a short distance, such as San Francisco, CA or a larger distance such as New York City, NY.

Unfortunately, these types of request logs are rarely published publicly, due to the privacy laws involved in releasing user identifiers, such as IP addresses. We therefore had to infer the types of properties found in such a workload and generated our own datasets based on these properties.

4.6 Summary

Providing users with low latency discovery of sensors in a wide-area network environment is a hard problem. Data sources and the links between them must be organised into an infrastructure that reduces the number of network paths traversed during discovery, in order to reduce network latency. The complexity of the problem is further increased with users requiring fresh and accurate results to their submitted requests. As a result of this, sensor metadata should be kept on the data source where updates are received, and the amount of caching and replication of metadata performed should be kept to a minimum.

In this chapter, we described how existing strategies for organising nodes and links into an overlay network fall short. While network-aware overlay networks provide low stretch, discovering data sources requires the traversal of multiple links, which results in a higher combined network path length. Semantic overlay networks provide an additional layer of abstraction constructed according to the content of data stored. Such structures provide high stretch as the organisation of data sources in the overlay network has no resemblance to the physical network regardless of the request workload.

We introduced the notion of a *workload-aware overlay network* (WAON), whereby the organisation of nodes, application-level links and users are determined according to the properties of the workload. The idea behind this overlay network is to use the data activity correlation approach from Chapter 3 for the construction of DAC clusters. These clusters are formed of data sources that have similar access patterns to one another and are thus, more likely to be able to handle subsequent requests amongst themselves.

The chapter presented Global Sensor Space, a global sensor discovery platform which realises the proposed WAON for data source discovery. We presented the online and distributed split-and-merge algorithm used by GSS. This algorithm ensures the self-organisation of DAC clusters and constructs the overlay network. It operates by decentralising split operations across clusters based on local knowledge, while centralising merge operations across clusters based on summarised global knowledge.

Our evaluation of the GSS approach showed a consistent improvement in user response time for all percentiles compared to an approach with a static structure. This improvement is larger under a geo-spatial locality workload than under a semantic-locality one, where GSS has a 45% median response time speed-up over a static overlay network structure. This shows how GSS adapts its overlay network according to the workload, providing a low stretch when the workload has high geo-spatial locality.

Despite these contributions, GSS and its underlying split-and-merge algorithm has limitations. Due to the requirement for a flexible query model and the trade-off between structured and unstructured overlay networks, network resources can be used inefficiently. Furthermore, the split-and-merge algorithm currently has a brute-force partitioning technique, which limits the scalability of GSS. Lastly, the evaluation has been limited due to restrictions on the latency model and workload.

Chapter 5

Data Placement: Geo-Dynamic Partitioning Middleware

User response time on popular online services is a major factor that determines user experience, which in turn can strongly affect a service's revenue stream. Facebook, Twitter and Google aim to provide their globally distributed user base with low latency access to stored application data. In adopting a multi-DC infrastructure, these online services can ensure that user requests are handled by a geographically-close DC, reducing the network path length between user and application data (see Section 2.1). However this is only applicable if the required data is stored at the closest DC, and therefore dependant on the data placement strategy adopted.

Existing strategies include master-slave full replication—adopted by Facebook—which can result in poor scalability, inefficient use of resources and prohibit small footprint DC deployments (see Section 2.2.2). Other placement strategies include data partitioning policies, which are commonly used in a single DC. Although the notion of partitioning application data can provide a variety of benefits such as increased throughput, performance and scalability (see Sections 2.3.1), these policies are not suitable for globally distributed deployments, suffering from poor latency performance over wide-area networks. An exception to this is geo-spatial partitioning, which partitions data according to a fixed geographical location attribute in the data, such as *city*, *county* or *country*. Partitions can then be dispersed across DCs in an attempt to reduce data access latencies. In general, how geographical locations are assigned to data and according to which heuristic is both challenging and application specific.

All current partitioning strategies are *static*, and therefore unable to handle dynamic workloads. Volley [ADJS10] attempts to address this by performing periodic batch processing of request logs to determine the placement of data (see Section 2.4.2). Such an approach is unable to handle the frequent shifts commonly found in workloads today (see Section 3.1), requires the periodic centralised gathering of request logs and relies on a processing cluster to perform batch computation. Other dynamic placement approaches such as SPAR [PES+10]

and SCHISM [CJZM10] focus on partitioning across machines in a single DC and are therefore not applicable to wide-area network settings.

In this chapter we focus on the development of a **data placement and discovery strategy** for multi-site deployments. We realise this strategy through Skyler, a geo-dynamic partitioning middleware for distributed data stores operating across multiple globally distributed DCs. The system utilises the current request workload to adapt both the data placement and discovery strategy in order to provide users with improved user response times. Skyler achieves this by using three techniques, outlined below and discussed further in this chapter.

Workload-aware data partitioning. Skyler partitions data that is stored across distributed data stores according to the DAC approach described in Chapter 3. User request frequencies for data items are stored as DFVs and used to identify similarities in access patterns. The same intuition applies: data items accessed frequently by the same set of users are correlated and therefore co-located on the same partition. DAC partitions provide a number of benefits, the most important being increased request locality. In coupling DAC with a novel online clustering algorithm, Skyler performs data partitioning *dynamically* and is therefore able to adapt to shifts in the workload.

Geo-aware partition placement. To improve user response times, application data must be placed at an appropriate DC. Rather than applying a placement strategy to each individual data item, Skyler performs placement of DAC partitions. This reduces the complexity of the problem and is a viable approach due to the high request locality in these partitions. Skyler migrates partitions between DCs using a simple weighted request mechanism based on the current request workload. In coupling workload-aware partitioning with geo-aware placement, Skyler can provide low data access latencies for a wide range of workloads, in addition to adapting to change (see Section 3.1).

Non-deterministic data discovery. Skyler is designed to be placed in-between the application layer and data store layer in order to intercept and redirect data requests to the appropriate partition according to a data discovery strategy. We explore a variety of these strategies ranging from more traditional, deterministic approaches to various non-deterministic approaches. Each of these strategies provides their own trade-off between latency and resource efficiency.

In this chapter, we present the system design and architecture for Skyler, describing its role in a single DC and global multi-DC infrastructure (Section 5.1). In Section 5.2, we introduce how the DAC approach can be used to develop a workload-aware data placement strategy by constructing DAC partitions. This is achieved using our *elastic partition algorithm* (EPA), a distributed clustering algorithm. After having appropriately placed data across partitions, Section 5.3 discusses how Skyler performs partition migrations between DCs, according to the current request workload. Section 5.4 discusses various strategies and their trade-offs for performing data discovery. We end the chapter with an evaluation of Skyler under various workloads and configurations in Section 5.5, followed by a discussion of this chapter in Section 5.6.

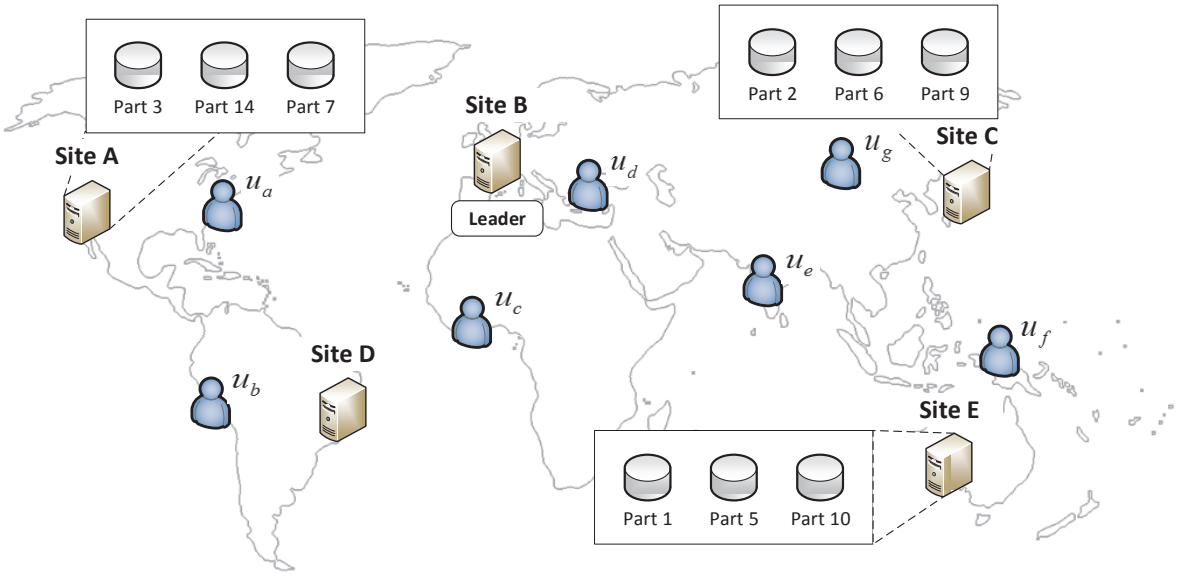


Figure 5.1: Global multi-DC deployment showing the separation of partitions across sites

5.1 System design

Skyler is a geo-dynamic partitioning middleware designed to be deployed across a multi-site wide-area networked infrastructure. It divides the available data into multiple DAC partitions, placing them across these sites. As discussed, an effective method to provide globally distributed users with low latency access to application data is to place data onto sites that are closest to users.

Therefore, for Skyler to achieve any significant latency improvements, it must be deployed over a *global* multi-site infrastructure. Figure 5.1 shows an example configuration for globally distributed data centres (Sites A–E), and end-users (u_a–u_g). Each site typically contains multiple application servers and data store servers storing data partitions. The figure shows an example configuration for partitions across these sites. With an effective strategy, this configuration will reduce the network latencies for users accessing these data partitions.

Existing WAN data placement strategies either place application data statically across sites—according to a crude approximation of the workload—or perform offline re-partitioning of data periodically (weeks or months). However the workloads of popular online services and applications are known to be both complex and dynamic. Multiple factors determine the interaction between user and data, and these factors continuously evolve (see Section 3.1). Therefore a data placement configuration that provides low latency results at one point in time can rapidly deteriorate and provide poor latency performance the next—irrespective of its initial quality.

For example, given the configuration in Figure 5.1, content located at Partition 3/Site A may initially be frequently accessed by user u_a. Its current placement ensures the user can access this data with low network latency because the site hosting this partition is geographically nearby. With passing time, the content at Partition 3 may begin to appeal to other users,

such as u_d and u_e, due to popularity and geo-spatial shifts. Intuitively its placement is no longer as effective, with users u_d and u_e having to perform high network latency requests. Other common factors of workloads that affect network latency include user mobility. For example, u_a still frequently accesses content at Partition 3, but is doing so from a different location that has a different closest site.

While replicating data across multiple sites can alleviate this problem, it comes with a new set of challenges. A system must ensure replicas are kept consistent according to the chosen consistency model, adding additional network overhead and complexity to the system. In many cases, however, replication is an essential part of the infrastructure due to the requirement of improved availability. Interestingly, the problem of placing replicas across sites in a partial replication model is orthogonal to the problem that we address in this chapter: the additional constraint is the need to minimise the network latency *between* replicas to ensure updates are made with high performance. We discuss replication and the various consistency models further in Section 2.3.2.

5.1.1 Middleware design

Today's large-scale online services typically structure their sites into a *three-tiered* architecture, consisting of the (1) front-end application (presentation layer), (2) application logic and servers (application layer) and (3) storage back-end (data layer). Application servers are arranged into clusters and handle incoming user requests from a front-end application, such as a friend request in an online social network or a search query in an online search engine. Most modern front-end applications are rich web applications written in JavaScript, with asynchronous AJAX communication [Gar05] to an application server on a nearby site. User requests are then handled according to the application logic, which may require the retrieval, updating or storage of data, resulting in subsequent requests issued to the data layer. Storage clusters consist of multiple distributed data stores, which persist data for the application logic to operate over.

The data layer can constitute a number of different types of data stores, depending on the application scenario. For example, a data layer may have multiple RDBMs such as MySQL [RT04], or NoSQL stores such as *column-based* store Apache Cassandra [LM10]), *key-value* store Dynamo [DHJ+07] and *document-based* store Apache Solr [SP11] (see Section 2.3).

Due to this versatility, we use the notion of a *data request* for the request or query to the applicable data store, which may involve the creation, reading, updating and deleting (CRUD operations) of application data. Although the architectures of online services and applications vary, such as with an additional caching or load-balancing layer [NFG+13], this basic tiered structure remains consistent.

Skyler is a partitioning middleware that is designed to be placed in-between the application and data layers. With this design, the requests previously issued from the application logic

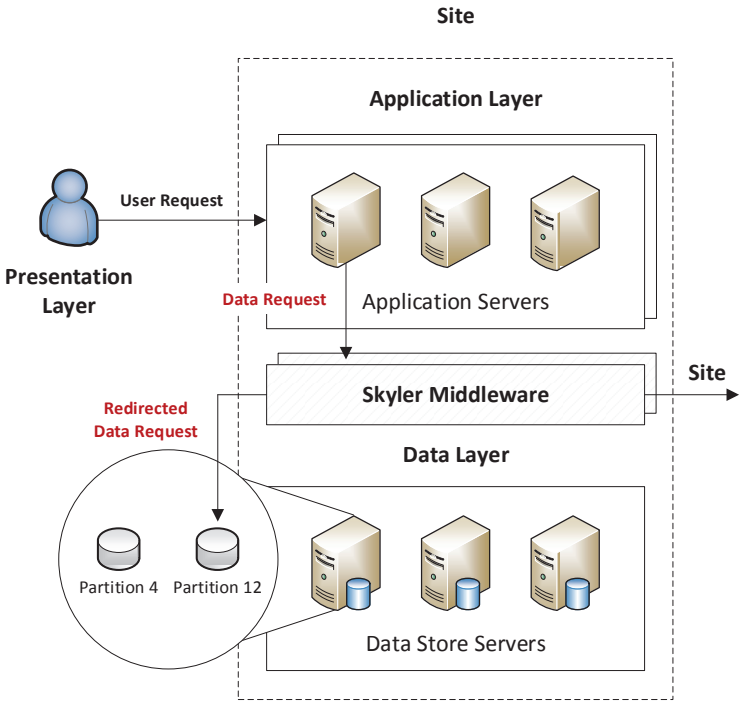


Figure 5.2: Placement of our middleware system in-between the front-end clusters (application layer) and data store clusters (data layer)

to data stores are now redirected through our middleware. This places the routing policy or *discovery strategy*—i.e. deciding which partitions and therefore which distributed data stores should receive a given request—in the hands of Skyler. Furthermore this allows for Skyler to collect the necessary statistics to maintain DFVs and AFVs used for partition construction, partition placement and alternative data discovery strategies. We discuss various strategies for performing discovery in Section 5.4.

Figure 5.2 shows the placement of the Skyler middleware in-between the application layer and data layer on a single site. The figure shows how the front-end application—as a result of a user interaction with the UI—directs a *user request* to an application server (typically by a load-balancer) at a site closest to the user. The application server then processes the user request according to the application logic, which in turn issues a *data request*. Skyler then intercepts this data request and puts its data discovery strategy into practice, selecting the partitions to redirect the data request to. Note that the figure also illustrates a data request being redirected to a different site.

There are various methods for determining a user's closest site, such as IP Anycast [EPS+98] and DNS Redirection [STA01]. IP Anycast is a technique whereby the same IP prefix is promoted from multiple sites. Is is then up to the network to decide which site to route a user request to, taking into account routing protocol costs. DNS Redirection is performed during the name resolution phase i.e. using the Domain Name Server (DNS) [MD88]. DNS provides a service on the Internet that maps domain names, such as *www.imperial.ac.uk*, to the IP addresses of machines. By modifying a DNS server, one can transparently redirect

users to the appropriate server without needing to add or modify the front-end application, software on the application server or network protocols.

5.1.2 Multi-site coordination

For Skyler to perform workload-aware partitioning and placement across multiple sites, it must have knowledge of the existence and status of all running instances. A Skyler deployment therefore requires a robust and flexible mechanism for multi-site coordination and general maintenance communication. We achieve this with Apache ZooKeeper [HKJR10], a reliable and scalable coordination system.

ZooKeeper provides a shared hierarchical namespace of data registers (called *znodes*) for the configuration, coordination, group membership and leader election and locking of distributed processes. This namespace is kept in-memory and replicated on all participating instances spread over multiple machines, providing it with strong fault-tolerant properties. A group or cluster of ZooKeeper instances is called an *ensemble*, made up of a single master or *leader* and multiple slaves. All write requests flow through the leader instance and are considered complete when a quorum of instances confirms the update. This is achieved through their Paxos-like protocol variant, ensuring strong consistency in addition to primary ordering of messages (see Section 2.3.2 for replication consistency models). Read requests can go to any instance, with the danger of retrieving stale copies of the data.

An important property offered by ZooKeeper is strong ordering of events. All updates issued to the service are totally ordered, achieved by leader transaction ID stamping. Read requests are also ordered with respect to updates—stamping them with the last processed update completed on the local ZK instance. This attention to order enables ZooKeeper to handle coordination in a large distributed system, avoiding the task of catching typical event race conditions found during node failure, joining and leaving amongst other scenarios.

Each Skyler instance holds a ZooKeeper component, creating a wide-area network ensemble made up of a single leader and multiple slaves. The selection of the leader site can be performed randomly or chosen to minimise the network latency between slaves. Coordination is achieved with each Skyler instance creating a register (or znode) in the namespace on start-up. With each instance broadcasting its presence, site location, partitions managed and status throughout the infrastructure. This information facilitates various other functionality such as the redirecting of data requests to partitions. Figure 5.1 shows an example multi-site deployment of Skyler with Site B elected as the leader of the ensemble.

5.2 Workload-aware data partitioning

Data partitioning across multiple data stores achieves improved scalability and other benefits (we have discussed data partitioning, various existing schemes and their benefits/trade-offs in Section 2.3.1). Existing single DC partitioning techniques are mostly static schemes with

the division of application data defined once, upon its creation. Other dynamic techniques such as with SPAR [PES+10] and SCHISM [CJZM10] are either application specific—e.g. focussing on optimising online social networks—or are offline solutions, requiring the re-computation of data partitioning (see Section 2.4.2).

In this section we describe a dynamic process for creating and maintaining data partitions, which we observe as a distributed data clustering problem. We present our *elastic partition algorithm*, an online distributed clustering algorithm that relies on our data activity correlation approach. Coupling DAC with a suitable distributed algorithm is what provides Skyler with its ability to adapt partitions according to the present workload. The motivations for co-locating data with similar access patterns on the same partition and in a multi-site wide-area networked infrastructure are outlined below:

Efficient partition placement. It establishes a high request locality coupling of data for an effective geo-aware partition placement strategy. A partition containing data with correlated access patterns ensures its migration as a whole is appropriate for all contained data members, and thus more efficient.

Reduced placement complexity. It reduces the complexity in performing data placement over a WAN, with an algorithm able to approximate the ideal placement for data items by abstracting the problem to data partitions. Placement of individual data items requires substantial CPU resources, thus limited to offline batch processing.

Improved efficiency and latency. It reduces the number of requests issued across multiple partitions due to its high request locality and access prediction properties. Request locality is important to have in a LAN and essential in a WAN because multi-partition and -site requests reduce network and resource efficiency in addition to increasing user response times [SC11].

However the majority of clustering approaches are centralised batch processing algorithms. In the following section, we provide an overview of the standard k-means clustering algorithm, which inspired us to develop an online distributed variation. We present the algorithm in Section 5.2.2.

5.2.1 Centralised K-Means clustering algorithm

Designing both an effective and efficient online distributed clustering algorithm is challenging. The algorithm must balance network communication and computation overheads with cluster quality. The algorithm we present, and which addresses this problem, is inspired by the popular *k-means* algorithm [Mac67, For65]. It is one of the simplest and most commonly used unsupervised learning algorithms that solves a familiar problem: to partition a set of N items into K groupings or clusters with high (prototype-based) *cohesion* within clusters and *separation* between clusters.

For our application scenario, clustering is performed on the data frequency vectors (DFV) of data items, denoted $\{\vec{x}_1, ..., \vec{x}_N\}$ (see Section 3.3). Cohesion is determined as the similarity between DFVs, and separation as the dissimilarity between DFVs in differing clusters.

Algorithm 3: Standard k-means algorithm [Mac67, For65].

Input: $\{\vec{x}_1, ..., \vec{x}_N\}, K$
Output: $\{c_1, ..., c_K\}$
1 $(\vec{s}_1, \vec{s}_2, ..., \vec{s}_K) \leftarrow$ SelectRandomSeeds($\{\vec{x}_1, ..., \vec{x}_N\}, K$)
2 **for** $k \leftarrow 1$ **to** K **do**
3 $\quad \vec{\mu}_k \leftarrow \vec{s}_k$
4 **while** *stopping criterion not been met* **do**
5 $\quad$ **for** $k \leftarrow 1$ **to** K **do**
6 $\quad\quad \vec{c}_k \leftarrow \{\}$
7 $\quad$ **for** $n \leftarrow 1$ **to** N **do**
8 $\quad\quad j \leftarrow argmin_l \, |\vec{\mu}_l - \vec{x}_n|$
9 $\quad\quad c_j \leftarrow c_j \cup \{\vec{x}_n\}$
10 $\quad$ **for** $k \leftarrow 1$ **to** K **do**
11 $\quad\quad \vec{\mu}_k \leftarrow \frac{1}{|c_k|} \sum_{\vec{x} \in c_k} \vec{x}$

There are numerous variations and implementations of the k-means algorithm [DM01,CGC01, Hua98,JD88], each with different objectives. The *objective function* for the standard k-means algorithm is to minimise the residual sum of squares (RSS), or the sum of squared errors (SSE), between items and their cluster centres. This is equivalent to the average squared distance when N is fixed [MRS08]. Cluster centres are defined as the mean or *centroid* of items in a cluster. Equation 5.1 shows the calculation for the centroid $\vec{\mu}$ of a given cluster c used in the algorithm.

$$\vec{\mu}(c) = \frac{1}{\parallel c \parallel} \sum_{\vec{x} \in c} \vec{x} \tag{5.1}$$

The RSS for a given cluster k is shown in Equation 5.2 with the combined final equation for RSS shown in Equation 5.3.

$$RSS_k = \sum_{\vec{x} \in c_k} |\vec{x} - \vec{\mu}(c_k)|^2 \tag{5.2}$$

$$RSS = \sum_{k=1}^{K} RSS_k \tag{5.3}$$

Algorithm 3 presents the standard k-means algorithm, illustrating how the algorithm executes as an iterative process: placing vectors $\{\vec{x}_1, ..., \vec{x}_N\}$ into the K clusters that are closest to their centroids $\{\vec{\mu}_1, ..., \vec{\mu}_K\}$. Line 1 computes the initial random centroid placement (later assigned on lines 2–3) to start the iterative process. This is one of the algorithm's weakest points and we discuss how our algorithm can mitigate its effects through the initial placement of data. The bulk of the algorithm is in lines 4–11, with the algorithms initial statement on line 6 emptying any previous cluster assignments. Lines 7–9 iterates through all item vectors

assigning them to the cluster that they have the shortest computed distance to. Line 8 is where this minimum distance centroid index j is found, and line 9 assigns the items to the actual cluster c_j. The algorithm then re-computes cluster centroids according to the latest assignments in line 11. In subsequent iterations, although clusters are re-initialised, the previous assigned centroids are used to place items progressively into clusters with reduced centroid distance.

The stopping condition for the algorithm is shown in line 4. It is left vague purposefully: for example, this condition could be the number of completed iterations, bounding the runtime of the algorithm but risking low quality clustering; other variations could use the RSS or its rate of change with a given threshold to determine termination. The most commonly used technique is to allow the algorithm to continue until there is no change in the assignment of items to clusters between iterations.

The standard k-means algorithm was designed to operate with the Euclidean distance function, which we can see in line 8 of Algorithm 3. Our data activity correlation approach, introduced in Chapter 3, utilises the cosine similarity with better performance. It produces quality clusters with sparse high-dimensional data and its unbiased nature. The spherical k-means algorithm (SPKM) [DM01, HFKB12] is a variation of the original k-means algorithm, which uses a different objective function: the cosine similarity measure. It instead *maximises the average cosine similarity* in all clusters. A further requirement of SPKM is that both data and centroid vectors are normalised.

5.2.2 Distributed Skyler clustering algorithm

Skyler uses an online distributed variation of the spherical k-means clustering algorithm to partition data, called the elastic partition algorithm (EPA). The benefit of being a variation is that the components of our DAC approach can be mapped onto those of a clustering algorithm: DAC partitions are clusters, data frequency vectors (DFV) are items, average frequency vectors (AFV) are cluster centroids and the cosine similarity is the distance or similarity measure. We describe the EPA algorithm in detail in the following section followed by an overview of its key differences to the k-means algorithm.

Elastic partition algorithm (EPA)

A difference between the clusters generated with a k-means algorithm and those behind our data partitioning algorithm is the flexibility in their group sizes. Clusters are able to be as imbalanced as the underlying dataset, whereas data partitions should remain as balanced as possible. The *elastic partition algorithm* (EPA) we present operates by *relaxing* the partition size constraints, increasing the amount of potential *flow* between partitions during the search for data item placement improvements.

The algorithm performs greedy data migrations from, say, partition c_a to partition c_b if and only if both partitions are sized within the lower and upper bounds set. This ensures that

Algorithm 4: Overview of the elastic partition algorithm shown from the perspective of a single site with K_s partitions (and K representing all partitions). This algorithm is run periodically given the set of AFVs ($\vec{\mu}_k$), the size of each partition (σ_k) and DFV selection parameter ϕ.

Input: $\{\vec{\mu}_1, ..., \vec{\mu}_K\}, \{\sigma_1, ..., \sigma_K\}, \phi$
1 **foreach** k *in* K_s **do**
2 $(\vec{x}_1, ..., \vec{x}_r) \leftarrow$ selectDFVs(c_k, ϕ)
3 **for** $n \leftarrow 1$ **to** r **do**
4 $current \leftarrow cosinesimilarity(\vec{\mu}_k, \vec{x}_n)$
5 $k_{remote} \leftarrow$ findBestAvailable$_l(cosinesimilarity(\vec{\mu}_l, \vec{x}_n), \sigma_l, current)$
6 **if** k_{remote} *found* **then**
7 doDataMigration$(\vec{x}_n, k, k_{remote})$
8 break
9 **if** *at least one migration for* k **then**
10 $\vec{\mu}_k \leftarrow \frac{1}{|c_s|} \sum_{\vec{x} \in c_s} \vec{x}$
11 propagateAFV$(\vec{\mu}_k)$

data is not migrated away from a partition that is too small; conversely data is not migrated to a partition that is too large.

Semi-fixed sized partitions, however, restrict the placement of data items. Data may, therefore, not be able to be placed in the partition that maximises the cosine similarity cohesion—our objective function. To ensure progressive improvement, our algorithm migrates items to the *best available partition*, taking into account their size and the cosine similarity improvement over their current placement. Although such migrations are sub-optimal, the overall configuration is still improved, ensuring constant progress toward a higher cohesion (assuming the rate of change in the workload is less than that of the algorithm).

Algorithm 4 gives an overview of the elastic partition algorithm that is executed on each Skyler instance distributed throughout sites. Each instances is responsible for managing K_s partitions. The algorithm is run periodically with the latest AFVs ($\vec{\mu}_k$) and sizes (σ_k) of all K partitions obtained locally and from the ZooKeeper component of Skyler. Furthermore the algorithm is given the fraction selection parameter ϕ, which is a set constant that determines the fraction of date items the algorithm processes at each iteration. The data item assignments for the kth cluster are denoted as c_k.

The algorithm starts by iterating through all the partitions maintained by the site (K_s), shown in line 1. For each partition k, it selects a fraction of the data frequency vectors (DFV) that have changed since the last round, according to ϕ (line 2). The cosine similarity for each selected DFV in their *current* placement in k is calculated (line 4), and compared with other partition centroids (or AFVs) that fulfil the size bounds, shown in line 5. If a higher cosine similarity partition k_{remote} is found, data item n is migrated from partition k to partition k_{remote}. Following a successful migration, the algorithm moves to the next selected DFV. The complexity of this algorithm for each Skyler instance on each iteration is $O(\beta K)$,

where β is the number of data items which have been accessed since the last iteration, and K the number of partitions in the infrastructure.

After iterating through all DFVs in a partition, it is important for the site to propagate the updated partition centroids to other sites, as shown in line 11. This ensures that other sites executing the algorithm have the latest data and able to make accurate migration decisions. The exchange of AFVs is achieved using ZooKeeper. Sites perform updates to znodes for each of the partitions that they maintain.

There are various factors that determine the effectiveness of this algorithm: ϕ, the rate at which the workload evolves, the partition minimum and maximum size bounds, and the propagation delay of local AFV changes to other sites. By only selecting DFVs that have changed since the last iteration, we can ensure that the algorithm's overhead is proportional to the number and diversity of requests in the workload.

Differences between EPA and K-Means

Although the k-means algorithm is similar to the elastic partition algorithm (EPA) presented in this section it also has a number of key differences, which we outline below:

Distributed datasets. Skyler distributes its DAC partitions across multiple sites, increasing the amount of computational parallelism that can be achieved during the clustering process. Each site can therefore maintain their own set of partitions and claim responsibility for the data items held within these partitions in addition to their in-memory stored DFVs.

Online clustering. The algorithm differs in that it is performed *online*. To achieve this we remove the stopping condition to the spherical k-means algorithm. With an offline clustering algorithm, the quality of clusters may deteriorate over time, due to a dynamic workload: DFVs and therefore AFVs—i.e. cluster centroids—change with each incoming user request.

Fixed size partitions. Partitions differ from clusters in that they hold a *semi-fixed* number of data items—i.e. a range with a minimum and maximum size. The data partitioning algorithm must therefore restrict the migration of data items to partitions that have reached their maximum size. This is important because smaller partitions in large-scale systems benefit from: (1) simpler partition management, such as performing backup and migration operations; (2) improved request performance, due to smaller data store instances and effective usage of the RAM cache; and (3) finer control over load-balancing.

Partition migration. Due to the distributed nature of our algorithm, we require robust and reliable coordination between sites. Coordination is required to share, for example, updated AFVs between sites. This information is essential for partitions to decide autonomously which data items to migrate in order to improve the current configuration. For example, migrating a data item held in a local partition to a remote partition due to a higher cosine similarity.

Centroid initialisation. A drawback to the k-means algorithm is its high variability in

results due to the poor initial random selection of centroids [YMZD04]. In Skyler, the initialisation of centroids—i.e. AFVs—depends on the initial placement of data items. Skyler initially places data at the site that is closest to the user generating it (a form of geo-spatial data partitioning): a reasonable estimate to its ideal placement when under a high geo-spatial locality workload. Furthermore given that our algorithm is online and the workload is dynamic, any bad initialisation will improve over time.

5.3 Geo-aware partition placement

Data partitioning schemes are typically deployed over a single site (or data centre) and therefore do not require a partition placement strategy. This is because placing a data partition at a machine as opposed to another machine at the same DC has minimal effect on the network latency of user requests.[1]

The dynamic data partitioning algorithm presented in Section 5.2.2 clusters data with similar user request patterns. It is unable to leverage the geographic location of sites with respect to users. In this section, we outline a partition placement strategy that is geo-aware, utilising user request statistics collected from the workload to make informed online decisions. The goal of this strategy is to actively reduce the network path lengths between users and the partitions that are frequently accessed by placing them at their closest site. This improves the user experience of the online application by reducing response times.

The intuition behind our partition placement strategy is to calculate a weighted geographical placement—i.e. a weighted mean calculation—for each partition, according to the source locations of request frequencies. The weighted placement and its partition can then be mapped to the closest site, determined with the *harvesine* distance [Sin84]. The harvesine formula provides the *great-circle* distance between two points on a sphere from their longitudes and latitudes. There are various methods to perform a weighted mean calculation. A common technique that is used for the placement of individual data items in Volley [ADJS10] is defined by Buss et al. as *progressive slerp* [BF01]. However, there are a number of inaccuracies to this method, such as the inconsistency in results depending on the order the points are calculated.

Another common technique used that does not depend on the ordering of points is the *euclidean mean* [MJ09]. In this method, the geographical coordinates (geodetic) are translated into 3D Cartesian coordinates (geocentric)[2] by approximating the earth with a sphere or ellipsoid, from which the weighted mean point can be calculated. This point is then translated back into geographical latitude and longitude coordinates to provide a weighted geographical mean location. This is the method which we use for our algorithm, as it is a simple technique to understand and provides accurate measurements.

To calculate the weights for source locations, the algorithm uses the same user request

[1] Partition placement at a DC may, however, be performed to balance request load.
[2] These are 3-dimensional points with the origin at the centre of the earth.

$^{Site}/_{Partition}$	$site_a$	$site_b$	$site_c$	$site_d$	$site_e$
$partition_2$	8	1	0	1	0
$partition_3$	0	3	0	3	4
$partition_7$	0	2	6	0	2

Table 5.1: Example of three site frequency vectors (SFV) with five sites

frequencies collected as part of our data activity correlation approach. Rather than organising these statistics into DFVs, we instead summarise this information into site frequency vectors (SFV). This is to provide a small fixed-sized vector for each partition that can be processed efficiently. SFVs hold the number of successful requests made to a partition from the various source sites. Table 5.1 shows an example for three partitions ($partition_2$, $partition_3$ and $partition_7$) in a deployment of five sites ($site_a$–$site_e$). The table illustrates each SFV for these partitions, showing the number of request that have originated from these sites.

An advantage of our algorithm over, for example, Volley's technique is that we only use the geographical locations of sites for request sources, rather than the locations of individual users. This reduces the computational complexity in performing the weighted mean calculation because the number of points that we must merge is proportional to the number of sites, not users. Skyler therefore assumes that each user in the system is mapped to their closest site—through techniques such as IP Anycast [EPS+98] and DNS Redirection [STA01]—and uses the site's location to represent them.

Algorithm 5 shows how the latitude and longitude geographical locations of all L sites ($\{g_1, ..., g_L\}$) are translated into geocentric Cartesian coordinates ($\{p_1, ..., p_L\}$). The algorithm begins by iterating through each site location (l) and converting its latitude and longitude coordinates from degrees to radians (lines 2–3). Lines 4–5 then perform the conversion from latitude and longitude into Cartesian coordinates, which are returned in line 7. The resulting Cartesian coordinates are used by the geo-aware partition placement algorithm. These coordinates are pre-computed offline due to the static geographical placement of sites.

Algorithm 6 shows the geo-aware partition placement algorithm executed periodically by each Skyler instance. The algorithm is given the SFVs for the K_s partitions being maintained ($\{f_1, ..., f_{K_s}\}$), in addition to the pre-computed Cartesian coordinates for the L sites ($\{p_1, ..., p_L\}$).

The algorithm begins by iterating over each partition and initialising the dimensions of a point and a total weight variable (line 1). Lines 3–7 show the weighted mean Cartesian coordinate being calculated for each of these partitions. This is achieved by iterating through the sites and multiplying the site point by the number of requests that have come from this site (lines 3–6). The final weighted point is then divided by the total weight—the sum of all requests issued to this partition (w_{total}).

Algorithm 5: Algorithm that translates the geographical latitude and longitude location of all L sites ($\{g_1, ..., g_L\}$ in degrees) into Cartesian coordinates ($\{p_1, ..., p_L\}$). These values are pre-computed and used later in the partition placement algorithm. Note that we omit the calculation for an ellipsoid.

Input: $\{g_1, ..., g_L\}$
Output: $\{p_1, ..., p_L\}$
1 **for** $l \leftarrow 1$ **to** L **do**
2 $\quad lat \leftarrow g_l.lat \times \pi/180$
3 $\quad lng \leftarrow g_l.lng \times \pi/180$
4 $\quad p_l.x \leftarrow \cos(lat) \times \cos(lng)$
5 $\quad p_l.y \leftarrow \cos(lat) \times \sin(lng)$
6 $\quad p_l.z \leftarrow \sin(lat)$
7 return $\{p_1, ..., p_L\}$

Algorithm 6: Geo-aware partition placement algorithm for a single Skyler instance with K_s partitions. The algorithm is given the SFV for partitions ($\{f_1, ..., f_{K_s}\}$) in a deployment with L sites, in addition to the pre-computed Cartesian coordinates for sites $\{p_1, ..., p_L\}$. Note that we omit the calculation for an ellipsoid.

Input: $\{f_1, ..., f_{K_s}\}, \{p_1, ..., p_L\}$
1 **foreach** *partition* k *in* K_s **do**
2 $\quad x \leftarrow 0\ y \leftarrow 0\ z \leftarrow 0\ w_{total} \leftarrow 0$
3 $\quad$ **foreach** *site* l *in* L **do**
4 $\quad\quad x \leftarrow p_l.x \times f_k[l]$
5 $\quad\quad y \leftarrow p_l.y \times f_k[l]$
6 $\quad\quad z \leftarrow p_l.z \times f_k[l]$
7 $\quad\quad w_{total} \leftarrow w_{total} + f_k[l]$
8 $\quad x \leftarrow x/w_{total}\ y \leftarrow y/w_{total}\ z \leftarrow z/w_{total}$
9 $\quad lng \leftarrow atan2(y, x) \times 180/\pi$
10 $\quad hyp \leftarrow \sqrt{(x^2 + x^2)}$
11 $\quad lat \leftarrow atan2(z, hyp) \times 180/\pi$
12 $\quad$ **if** *closest site to* (lat, lng) *differs* **then**
13 $\quad\quad$ migratePartition(k)

The final part of the algorithm is to translate the mean Cartesian point back to latitude and longitude in order to accurately compare it with the geographical location of sites (lines 9–10). If the closest site to this computed latitude and longitude is different, Skyler initiates the partition migration protocol. The complexity of this algorithm for each iteration is $O(K_s L)$, where K_s is the number of partitions and L the number of sites in the system. Each partition requires its mean point to be calculated using the location and weights of sites.

Figure 5.3 illustrates the partition placement process with an example configuration and weights. The figure shows how the weighted geographical mean of three partitions is calculated using the site frequency vectors defined in Table 5.1. Sites A-E are given a latitude and longitude location, which when combined with weights, is used to place partitions on the

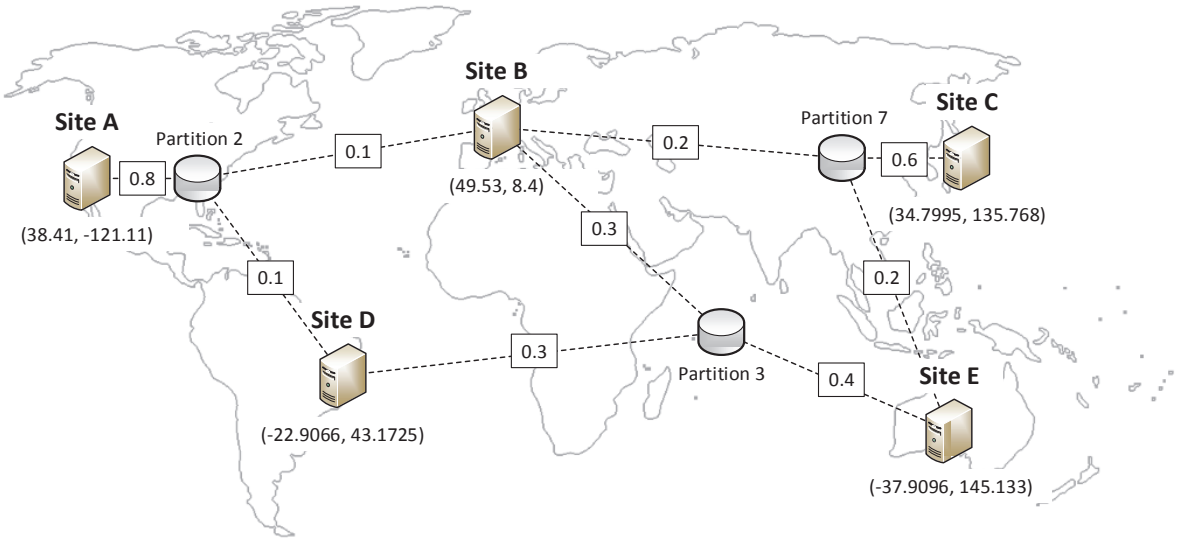


Figure 5.3: Example weighted geographical mean calculation for three data partitions and five sites

earth. The site closest to the computed placement of partitions is the one that stores and maintains it. Note that it is possible for a site to store a partition that has not redirected any requests to it. For example, if a partition had equal weights to Sites D and C, its placement would either be at Site B or Site A.[3]

In this section, we have described the strategy used by Skyler to place data partitions at sites for the purpose of reducing user response times. However, dynamic partition placement is only possible with access to an *online migration* (OLM) mechanism, which provides seamless data migrations between sites whilst concurrently handling user requests. There are various existing solutions, tools and utilities to perform this operation [LAW02, Mad97],[4,5] all of which follow a similar protocol.

The technique for performing online partition migration is to create a temporary slave or replica partition at the chosen destination site. The slave is then kept consistent with the master by transferring all of its data across, in addition to any further updates i.e. create, update or delete requests (see Section 2.3.2). Upon reaching a consistent state, the master is brought down and the slave is promoted to master. One method for achieving this slave/master coordination is to use the ephemeral znode in ZooKeeper. An ephemeral znode ensures that all of the sites are told when a master fails, cueing the next slave to promote itself. An optimisation to this process is to transfer the bulk of the data from master to slave before executing a consistency protocol. This ensures the latency penalties involved in a consistency protocol are kept to a minimum.

[3]Note that the earth wraps around, and thus Site A may be closer to the midpoint between Sites D and C.
[4]Online Shard Migration (OLM) at Facebook, `http://www.percona.com/live/mysql-conference-2013/sessions/online-shard-migration-olm-facebook`, last accessed: 25/5/2013
[5]Veritas Replicator, `http://www.symantec.com/en/uk/replicator`, last accessed: 25/5/2013

5.4 Non-deterministic data discovery

User actions performed by the front-end application are translated to user requests, which are submitted to an application server typically located at the closest site to the user. The application server processes this request according to the application logic, which may require subsequent data requests to be issued to the back-end data storage (see Section 5.1.1). With a large-scale distributed data store, application data is divided into data partitions stored at multiple physical machines. The difficulty therefore lies in forwarding this data request to the appropriate partitions, ensuring both network efficiency and low network latency. The network latency is depicted by the *total* network path length in routing data requests between sites. The strategy for forwarding requests in a multi-site infrastructure given a placement configuration, is what we refer to as the data discovery problem (see Section 2.1).

In the following sections, we explore various data discovery strategies that can be used in a WAN and discuss their trade-offs. The first three strategies are variations of existing techniques and include data partitioning functions, forwarding/lookup tables and broadcast discovery (closest site). The remaining strategies are Skyler specific and utilise the user request statistics to make informed request forwarding decisions. These include broadcast discovery (best partition), greedy DAC discovery and weighted DAC discovery.

5.4.1 Data partitioning function

Traditional data partitioning technique divide data according to a *function* and a given partitioning key [RG00, GDQ92, KLL97]. The application logic uses the function and partitioning key to compute the relevant partition identifier, which the data request should be forwarded to (see Sections 2.3.1 and 2.4.1). Partition identifiers are then mapped to machines using a static partition *lookup table*.

The advantage of this is that it is simple, scalable and efficient when the partitioning key is given in the request. The disadvantage is that a data request must contain the partitioning key for the function to be computed. Given a data request without the partitioning key, the application logic must send the request to *all* partitions to ensure application data is retrieved i.e. *broadcast* the request. This is unacceptable over a WAN in which network latencies between DCs are high. To ensure most data requests contain the partitioning key, there is added complexity to the application software. Another disadvantage to this approach is that the placement and discovery use the same function. As function-based strategies for data partitioning are *static*, so are their counterpart data discovery strategies. This technique is unable to handle, for example, arbitrary re-partitioning of application data.

5.4.2 Forwarding/lookup table

An alternative strategy for performing data discovery in both a static and dynamic environment is to use a *global forwarding* or lookup table [TCJM12]. Such a table is an index that

maps a chosen attribute in the data to partition identifiers. In a LAN deployment, this can be stored by a standalone server and used as a forwarding service. For fault-tolerance and to minimise the single service bottleneck, the index can be replicated across machines. In a WAN deployment, the index stores global mappings and therefore must be replicated across sites to ensure any application servers can forward their data requests successfully to the correct site and partition. Twitter is an example of a service that uses forwarding tables in their distributed data store infrastructure.[6]

The advantage to this strategy is that is can support the lookup of dynamically placed data. Upon moving application data from one partition to another, the index is updated and changes are propagated to all replicas. This strategy can also index *multiple* attributes, adding flexibility to the discovery process.

A disadvantage of this strategy is the need to maintain consistent index replicas, adding complexity and overhead. This can become a bottleneck for a *dynamic* placement strategy that performs frequent migrations. A further disadvantage is that indices can become large, storing the mappings for millions or even billions of application data rows. This becomes even larger when dealing with a multi-attribute index. Lastly, searching for the placement of application data without using the indexed attribute requires the broadcasting of the request to all sites and partitions.

As a middleware, global forwarding tables can easily be implemented in Skyler. Each instance stores a replica of the index and any update to the placement of data, e.g. due to data migrations, are propagated between the sites. Although this can reduce the throughput of operations such as data migrations, neither the dynamic data partitioning or partition placement mechanisms are on the critical path.

The data partitioning function and forwarding table strategies discussed above can be categorised as *deterministic* discovery approaches, for which the placement of data can be determined perfectly. While this can reduce the number of sites and partitions visited and improve efficiency, it either enforces static placement or adds additional overhead for maintaining a lookup table across sites. The following strategies that we discuss are *non-deterministic*, and thus remove these overheads. Instead, these informed strategies attempt to determine the placement of data in a WAN without certainty, balancing the trade-off between network efficiency and performance.

5.4.3 Broadcast discovery (Closest Site)

The simplest strategy for forwarding data requests in a WAN, is to broadcast data requests to all sites. Requests are first forwarded from the user to a single *initial site*. On failing to serve the request, the initial site broadcasts a data request to all other sites in the infrastructure. We discuss two variations of the broadcast discovery strategy, which differ in how they select the initial site. In this strategy we select the initial site as the user's geographically *closest*

[6] Gizzard (Twitter), `https://github.com/twitter/gizzard`, last accessed: 11/5/2013

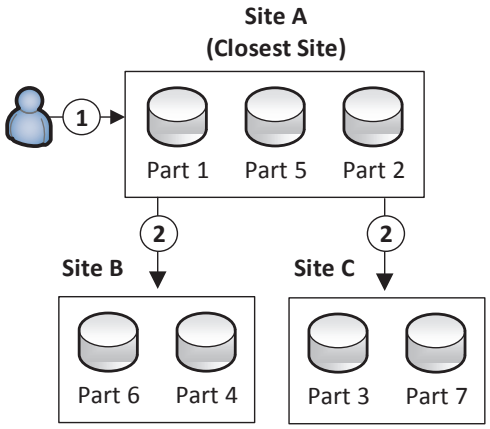


Figure 5.4: Example of a broadcast discovery (CS) strategy with Partitions 1-7 on Sites A–C, Site A is the user's initial site

site (CS). This is the standard policy and one that is used in the data partitioning function and forwarding table strategies.

Figure 5.4 shows the steps in the broadcast process with an example. In the example, the front-end application of a user sends a user request to its closest site (Site A), shown as step 1 in the figure. The initial site performs a *local search*—i.e. a search amongst the partitions stored at the site—for the content using either a local lookup table or by broadcasting the request to all of the machines in the site. A local broadcast is performed if, for example, the request is for an attribute that is not indexed (see Section 5.4.2). On success, the application data is updated, added, deleted or retrieved according to the application logic and returned to the user. On failure, the data request is broadcast by the initial site to all the other sites in the global infrastructure, shown as step 2 in the figure.

A major advantage of such a strategy is its simplicity, eradicating the need for a distributed lookup service altogether and the network overhead for keeping tables consistent. The disadvantage to the strategy is the high network latency of requests and network inefficiency when local searches at the initial site fail. However, the strategy can work well in practice if local searches at the initial site succeed frequently. Successful local searches can be maximised by aligning the data placement strategy with the initial site strategy, such as geo-spatial partitioning with the closest site (CS) strategy. Local searches can be optimised further by heavily replicating application data across sites.

5.4.4 Broadcast discovery (Best Partition)

In the previous strategy, the initial site selection was the closest site (CS) to the user. In this strategy we select a user's initial site according to the success rate of previously issued requests. Each user maintains a result frequency vector (RFV) (see Section 4.2.1), with dimensions representing partitions and the frequencies stored representing the number of successful requests returned by each partition. Front-end applications use RFVs to make informed request forwarding decisions: forwarding requests to the site storing their *best*

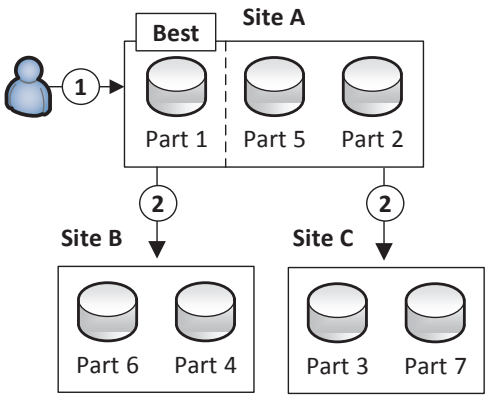


Figure 5.5: Example of a broadcast discovery (BP) strategy with Partitions 1-7 on Sites A–C, Partition 1 is the user's best partition and Site A is the user's initial site

partition (BP)—the partition that has most successfully responded to requests within the given window size. For example, user u_a performs 7 requests out of which 3 are handled by Partition 2 (stored on Site B), 2 by Partition 1 (stored on Site A) and another 2 by Partition 3 (also stored on Site A). In this example, the best partition for user u_a is Partition 2, which is stored on Site B.

The broadcast discovery strategy uses the *best partition* for its initial site heuristic. The strategy and its steps are illustrated in Figure 5.5, where the users best partition is Partition 1 stored on Site A. First, the user forwards its request to the site storing the best partition, shown as step 1. On success, the request is handled and a response is given to the user. On failure, the site broadcasts the data request to all other sites in the infrastructure, shown as step 2 in the figure.

The advantage of this strategy is that it can provide low network latencies and network efficiency when users exhibit temporal request locality i.e. frequently request the same data items. The strategy also removes the need for a global lookup table, simplifying the architecture and reducing maintenance overheads. The disadvantage of this strategy is the high network latencies when users lack temporal request locality. This is because the initial site may be geographically far away from the user. Other disadvantages discussed in the previous broadcast strategy also apply to this strategy (see Section 5.4.3).

5.4.5 Greedy DAC discovery

An alternative data discovery strategy is to perform informed request forwarding in the infrastructure, at each step. This differs from the broadcast discovery (Best Partition) strategy, which only performs informed request forwarding on the first step i.e. forwarding the request to the user's initial site. The idea behind this strategy is to predict the site in which the required data is stored based on past access patterns.

Each Skyler instance manages an average frequency vectors (AFV) for each stored data partition, providing a snapshot of the workload handled by the partition. With this, Skyler

Algorithm 7: Algorithm for constructing the partition ranking of a root partition

Input: AFVs of all partitions P: $\{\vec{p_1}, ..., p_{|\vec{P}|-1}\}$, AFV of root partition: $\vec{p_r}$
Output: Partition ranking $part_{rank}$ for root partition $\vec{p_r}$

```
1 part_rank ← {}
2 foreach partition p_x in all partitions P do
3     part_rank ← add(cosinesimilarity(p_x, p_r), part_rank)
4 sort(part_rank)
```

Algorithm 8: Algorithm for constructing the greedy site ranking of a root partition

Input: Partition ranking $part_{rank}$: $\{\vec{p_1}, ..., p_{|\vec{P}|-1}\}$ for root partition: $\vec{p_r}$
Output: Greedy site ranking $greedysite_{rank}$ for root partition $\vec{p_r}$

```
1 greedysite_rank ← {}
2 foreach partition p_x in partition ranking part_rank do
3     site ← getSite(p_x)
4     if site ∉ greedysite_rank then
5         greedysite_rank ← add(site, greedysite_rank)
```

can construct a unique *partition ranking* for each partition held, setting themselves as the *root partition*. Constructing a partition ranking involves computing the cosine similarity between the AFV of the root partition and all the other partitions. The cosine similarity measurements between partitions are then ranked, with a higher rank depicting a higher similarity.

Algorithm 7 shows the process of constructing a partition ranking $part_{rank}$ for root partition $\vec{p_r}$. The algorithm begins with the initialisation of the $part_{rank}$ sorted list in line 1. Each AVF of a partition, $\vec{p_x}$, is compared with the root partition $\vec{p_r}$ using the cosine similarity. These measurements are stored in the partition ranking, shown in line 3. The algorithm finishes after sorting the partition ranking in line 4.

Skyler uses the partition ranking to construct a *greedy site ranking*, used to determine the likelihood of discovering data at each site given that the root partition is the user's best partition (see Section 5.4.4). A site ranking is constructed by iteratively selecting the highest ranked partition in a partition ranking and determining its site. Algorithm 8 shows the process for constructing the greedy site ranking $greedysite_{rank}$ of root partition $\vec{p_r}$, given the partition ranking $part_{rank}$. The algorithm begins with the initialisation of the $greedysite_{rank}$ sorted list in line 1. For each AVF $\vec{p_x}$ in the *sorted* partition ranking, the partition's site is obtained (line 3). If the site has not previously been appended to the greedy site ranking, it is appended to the end of the list (line 4–5).

The strategy utilises the greedy site ranking to determine the ordering in which sites should be forwarded requests. Rather than forwarding requests to all the sites concurrently, requests are forwarded to sites sequentially. Figure 5.6 shows the greedy DAC discovery strategy with an infrastructure consisting of Partitions 1–7 stored on Sites A–C. In this example, the user request is first forwarded to the initial site: the site storing the user's *best partition*,

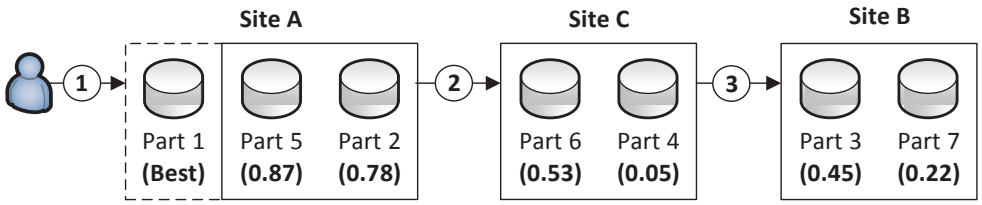


Figure 5.6: Example of the greedy DAC data discovery strategy with best partition (Partition 1) stored on Site A and a greedy site ordering of Site C and Site B

Algorithm 9: Algorithm for constructing the weighted site ranking of a root partition

Input: Partition ranking $part_{rank}$: $\{\vec{p_1}, ..., p_{|\vec{P}|-1}\}$ for root partition: $\vec{p_r}$
Output: Weighted site ranking $weightedsite_{rank}$ for root partition $\vec{p_r}$

```
1 weightedsite_rank ← {}
2 partcount ← {}
3 foreach partition p_x in partition ranking part_rank do
4 |   site ← getSite(p_x)
5 |   weightedsite_rank[site] ← weightedsite_rank[site] + cosinesimilarity(p_x, p_r)
6 |   partcount[site] ++
7 foreach site s_x in weighted site ranking weightedsite_rank do
8 |   weightedsite_rank[site] ← weightedsite_rank[site]/partcount[site]
9 sort(weightedsite_rank)
```

which is Partition 1. The figure shows the greedy ranking of sites based on the similarity measurements of partitions. Site C is the next site to be forwarded the request after Site A (step 2) due to Partition 6 having a cosine similarity of 0.53, which is higher than any other in the partition ranking. Note that Site C is next regardless of it storing Partition 4, which has the lowest similarity measurement of 0.05. Lastly, should Site C fail to provide the required data for the request, the request is forwarded to Site B (step 3).

5.4.6 Weighted DAC discovery

A variation to the greedy DAC discovery strategy is to construct a *weighted site ranking* from a partition ranking. Rather than greedily choosing sites according to their highest ranked partition, weighted site rankings are determined according to the average similarity of partitions in sites. This ensures the ordering of sites is not effected by similarity calculations that are outliers.

Algorithm 9 shows the process of constructing the weighted site ranking $weightedsite_{rank}$ for root partition $\vec{p_r}$, given the partition ranking $part_{rank}$. The algorithm begins with the initialisation of the $weightedsite_{rank}$ sorted list and the vector counter *partcount* in lines 1–2. For each AVF $\vec{p_x}$ in the partition ranking, the partition's site is obtained in line 4, and used to store the sum of cosine similarities in $weightedsite_{rank}[site]$. Line 6 shows the $partcount[site]$ variable incremented to record the number of partitions in each site. Once the iteration is complete, the algorithm iterates over the sites and divides their cosine

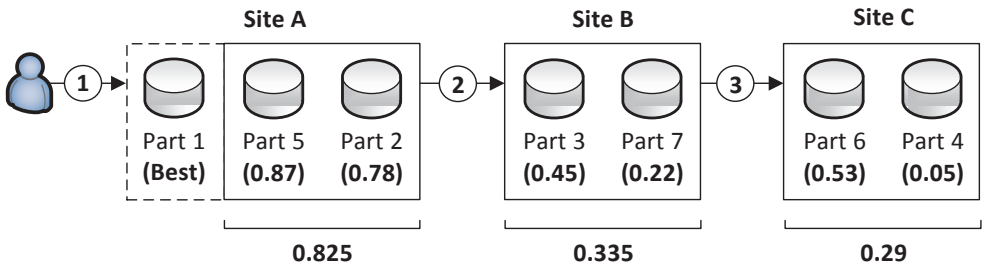


Figure 5.7: Example of the weighted DAC data discovery strategy with best partition (Partition 1) stored on Site A and a weighted site ordering of Site B and Site C

similarity total by the number of partitions to provide an average weight per site (lines 7–8). The algorithm ends after sorting the weighted site ranking, shown in line 9.

Figure 5.7 shows the weighted DAC discovery strategy, with the same set-up as Figure 5.6: an infrastructure consisting of Partitions 1–7 stored on Sites A–C. The figure shows ordering of sites in which to forward requests based on the average similarity of partitions in each site. As a result of this process, the ordering of sites differs to that of the greedy strategy with Site B given a higher ranking to Site C. In this example, the user request is first forwarded to the initial site: the site storing the user's *best partition*, which is Partition 1. Site B is the next site to be forwarded the request after Site A (step 2) due to it having a weighted cosine similarity of 0.335, which is higher than any other in the greedy site ranking. Lastly, should Site B fail to provide the required data for the request, the request is forwarded to Site C (step 3).

Both the greedy and weighted DAC discovery strategies have similarities to the informed data discovery strategies described in Section 2.5.3. Intelligent search [KGZY02], for example, utilises the history of previously successful requests to guide requests over an unstructured overlay network. Forwarding is, however, performed on a per-request basis, which differs from our per-partition strategy. Because of this, intelligent search compares the *content* (i.e. keywords) of the request to past requests that were successful.

There is a clear trade-off between broadcast and sequential discovery strategies: broadcasting requests to all sites reduces network efficiency but achieves lower access latency, while naïve sequential strategies does the reverse. However, should a sequential strategy predict the placement of data accurately, it can be both network efficient and a low access latency strategy. A broadcast strategy, such as those in Sections 5.4.3 and 5.4.4, can be represented as a *shallow forwarding tree* with a depth of one. Whereas a sequential strategy, such as those in Sections 5.4.5 and 5.4.6, as a *deep forwarding tree* with a depth equal to the total number of sites. Although we do not discuss this further, an interesting alternative to these two general techniques is to employ a hybrid approach. For example, rather than constructing extremely shallow or deep forwarding trees, one could construct trees of various depths. Changing the shape of forwarding trees provides Skyler with the ability to leverage the trade-off between network efficiency and latency.

5.5 Evaluation

In this section, we empirically evaluate the SKYLER middleware system and its various components. Although we provide results highlighting the various trade-offs, our evaluation focuses on the performance metric that has been common throughout this thesis: network latency. We investigate SKYLER under a range of workloads, both static and dynamic, identifying the conditions in which our strategies excel and those that provide moderate benefits over existing placement approaches.

5.5.1 Evaluation methodology

We perform the evaluation of the SKYLER middleware and its three major components in simulation. The simulator used is a custom built event-based simulator in Java, designed to mimic the processing of requests submitted by globally distributed users. Each SKYLER instance in the simulator manages their own data partitions and performs the required runtime request logging. All partitions and data item identifiers are stored in-memory for improved execution time.

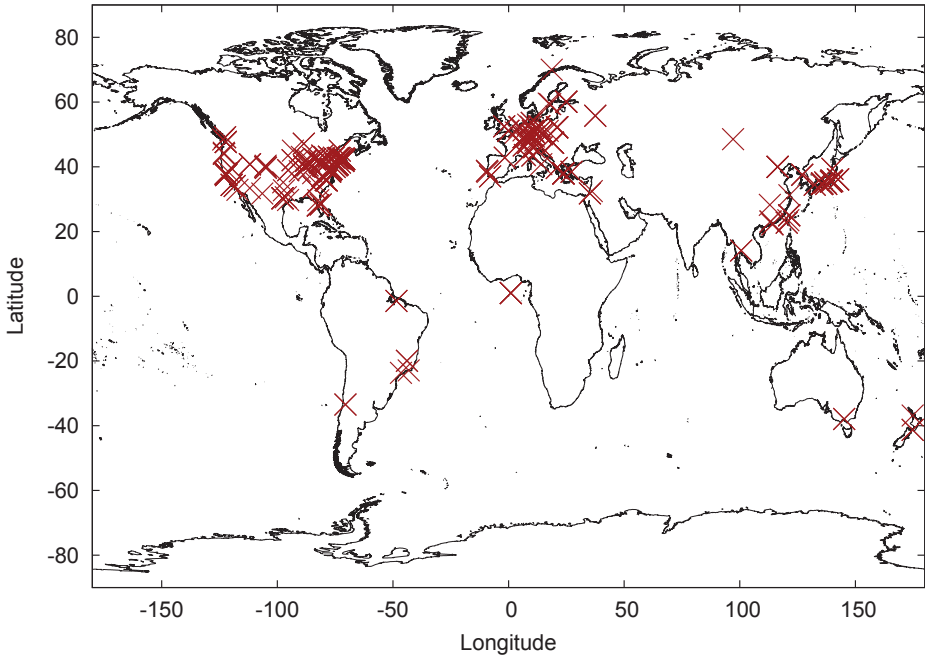
Users requests are modelled as *single-get* requests (unless otherwise specified) involving the retrieval of a single data item stored in the infrastructure. This has minimal effect on the results as a multi-get request is equivalent to multiple single-get requests performed concurrently. Many of the strategies we compare against require the request to contain a partitioning key or indexed attribute for a successful discovery, which we therefore assume is always given.

The main objective of our simulation is to measure the network latency in responding to a user request with various data placement and discovery strategies. It is therefore important that we have an accurate reflection of the global distribution of nodes and the latencies between them. This is achieved with a latency model dataset: an $n \times n$ latency matrix, which is constructed by collecting and processing multiple real-world Internet host measurements.

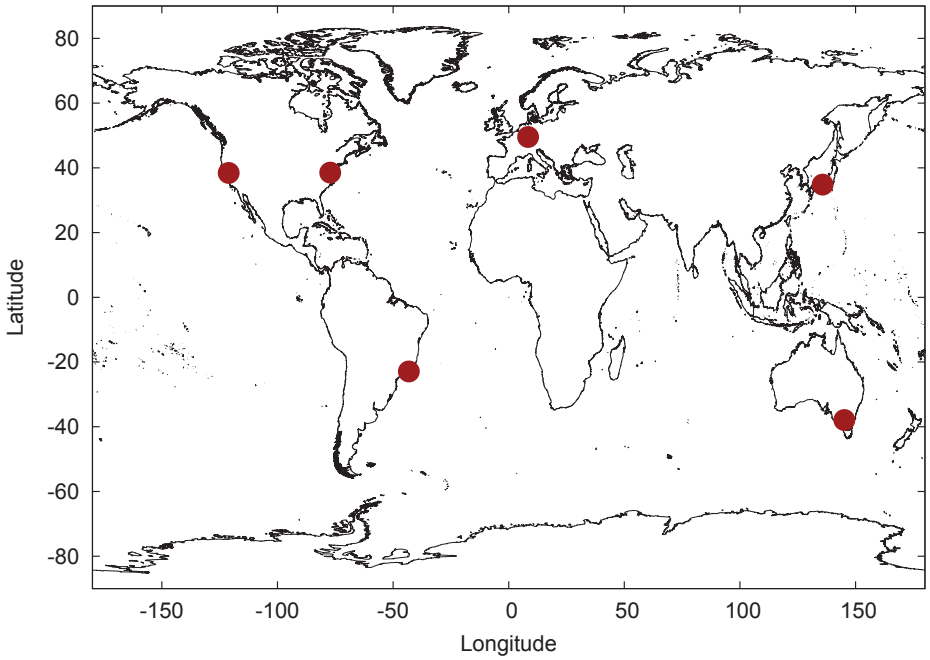
Existing latency model datasets are either too small[7] or incomplete, and hence ill-equipped for our needs. We therefore perform our own collection of a dataset by deploying a custom built network latency measurement service on the PlanetLab (PL) platform (see Section 4.4.1). Each available PL node runs this service, and is instructed by a centralised logging server to perform ping test operations to various other hosts. This measures the round-trip time (RTT) between two hosts by sending ICMP echo request packets and awaiting a response. The logging server constructs a complete latency matrix by requesting hosts to perform measurements where there are empty cells in the matrix.

The latency dataset used throughout the evaluation was collected over a 24 hour period. Although there were 465 PL nodes involved in the collection phase, the final complete matrix consists of 342 PL nodes. This is because PlanetLab is a shared resource with nodes

[7]PlanetLab Median Dataset, `http://www.eecs.harvard.edu/~syrah/nc/`, last accessed: 10/10/2011



(a) Geographical distribution of users



(b) Geographical distribution of data centres

Figure 5.8: Geographical distribution of the PL nodes in the latency model dataset, and their division into users and data centres

frequently failing, under variable load and network utilisation, which affects the dataset collection process.

For the evaluation of a globally distributed infrastructure, the workloads handled must have a variety of geo-spatial properties. We therefore obtain the geographical locations of PL nodes—provided by the PlanetLab API service—and calculate their distance matrix using the harvesine formula [Sin84]. This matrix is then used by our static and dynamic workload generators, which we outline further in Sections 5.5.2 and 5.5.3, respectively.

Figure 5.8 illustrates the geographical distribution of the PL nodes in our latency model dataset. Figure 5.8(a) shows the 336 nodes that are used to emulate users. Although the distribution of users is global, it is also skewed because the East Coast US, Central Europe and Japan are more densely populated. This skew is a limitation of the evaluation, which we discuss further in Section 5.6. Our evaluation is performed using 500 *virtual* users, which all map to one of the PL nodes. Figure 5.8(b) shows the distribution of 6 PL nodes, which we define as our data centres. We have purposely selected key regions for the placement of data centres to maximise the user catchment area, as per any real multi-site deployment.

Unfortunately there is a lack of publicly available global-scale request log datasets that refer to large-scale infrastructures. While there are a number of existing workload generators, none considers the geographical distribution of users and their relationship toward the data requested. Instead generators such as YCSB [Coo10], SearchGen [LLSG07] GlobeTraff [KXP12] and more traditional benchmarks such as TPC-W[8] and TPC-E[9] focus on modelling popularity distributions of data, read-write request ratios and the inter-arrival rates of requests.

We therefore evaluate aspects of the Skyler system with two workload generators that we have developed. Our *static workload generator* allows for the fine-tuning of geo-spatial locality properties, while the *dynamic workload generator* models the rate of shift in content popularity and geo-spatial locality. All experimental runs of our simulators are performed using the 64-bit 1.7.0_21 Java JVM build on a 64-bit machine consisting of an Intel Core i7 1.73Ghz processor with 8 GB of RAM.

5.5.2 Static simulation

The latency of a multi-site infrastructure depends on (1) the data placement and discovery strategy adopted; and (2) the geo-spatial locality properties of the workload. For example, users may request data that is semantically related to their geographical locations, providing an opportunity for an appropriate strategy to serve their requests from nearby data centres. Conversely a workload may be inversely related to their geographical location, i.e. users tend to request data that is semantically related to a distant location.

Popular online services handle a wide range of workloads with varying properties (see Sec-

[8]TCP-W, `http://www.tpc.org/tpcw/`, last accessed: 12/5/2013
[9]TPC-E, `http://www.tpc.org/tpce/default.asp`, last accessed: 12/5/2013

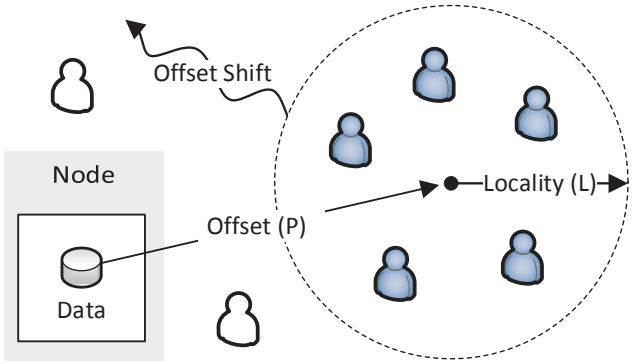


Figure 5.9: Static and dynamic workload model used for evaluating Skyler

tion 3.1). While many location-based social networks (LBSN) such as Gowalla and BrightKite have high geo-spatial locality workloads, others such as Twitter have lower geo-spatial locality properties. In this section, we evaluate Skyler against a variety of existing strategies using a parametrised static workload generator.

The generator is designed to produce a stream of requests that are issued from a user for an individual data item. It operates based on a given (1) geographically distributed users and multi-site infrastructure; (2) geographical distance model between nodes; (3) locality parameter; and (4) offset parameter.

Locality. Each data item is mapped to a PL node location and given an area of influence around it, depicting the users that are interested in this data. The locality parameter L in our workload model defines the radius of this area, with high locality resulting in a small area and a low locality parameter a large area. We fix the distances of this parameter to 20 km for a locality of 1.0, ranging to an extreme of 16,000 km for a locality of 0.0. These distances are chosen to ensure that a low locality can encompass an area close to the entire earth (approximately half of the circumference of earth i.e. 40,075 km).

Offset. The area of influence given to data items can also be displaced, with its centre or origin moved from the chosen PL node's location. The offset parameter P provides the amount of displacement, which, similar to locality, has a fixed distance of 20 km for a offset of 0.0, and 16,000 km for an extreme value of 1.0. A high offset places the area of influence of a data item far across the earth, whereas a low offset places it at the location of its chosen PL node.

Figure 5.9 conveys the intuition behind the locality and offset parameters. Data items are first placed at the location of a random PL node, with an area of influence placed around this location. The radius of this area is depicted by the locality parameter, as shown in the figure, and the centre (or origin) of this area is displaced according to the offset parameter.

The workload generator pre-computes a set of users for each data item using the location and radius of their areas of influence. With this, the generator can perform a number of simple steps to produce a request: (1) a data item is uniformly selected from the set of all data items throughout the system; (2) its set of interested users is retrieved and a single user

is selected uniformly; and (3) a request is generated by this user for the data item.

Experimental set-up

Experimental runs performed with the static workload generator consist of 500 virtualised users submitting 100,000 requests. These requests are for 2000 different data items, stored across 6 globally distributed data centres. Experiments are also performed using a data frequency vector (DFV) window size of 200 ($window_{DFV} = 200$) and site frequency vector (SFV) window size of 200 ($window_{SFV} = 200$). These values were defined through empirical testing with the same experimental set-up and a static workload, exhibiting both stability and adaptability. Data items are divided into 100 partitions of size 20, with the elastic partition algorithm given a 15% expansion/reduction. This results in a maximum partition size of 24, and minimum partition size of 16. Experiments with different parameters to the ones we have described above are explicitly mentioned.

In our evaluation using a static workload, we compare five different data placement and discovery strategies. Although discussed throughout the thesis, we summarise these strategies below:

Geo-spatial partitioning. In this strategy we simulate *perfect* geo-spatial partitioning, in which data items are placed at sites which are geographically closest to their initial PL node location. The placement remains static throughout the experimental run. In practice, performing moderately accurate geo-spatial partitioning is challenging, with developers estimating the ideal location for data based on crude heuristics (see Section 2.4.1).

Hash-based partitioning. Although hash-based partitioning in a wide-area network is unlikely to be performed in practice, it provides us with a lower performance bound to compare with. In this scenario, application data is randomly dispersed across sites, remaining static throughout the experiment. User requests are forwarded to their closest site.

Skyler (CS). This strategy uses our Skyler middleware with the closest site (CS) heuristic. The advantages of this are its simplicity and ability to select an initial site using request statistics already collected. Upon failure on the initial site, the request is broadcasted to all other sites. The initial placement of data items for both Skyler variations, which we outline here, follows the same policy as geo-spatial partitioning.

Skyler (BP). A variation of the above strategy is to adopt the *best partition* heuristic. In this strategy the initial site is the site that stores the partition that has successfully handled the most requests. The result frequency vector (RFV) window size used throughout the evaluation is 40 ($window_{RFV} = 40$), providing both stability and adaptability in choosing a user's initial site.

Full replication. The above strategies do not replicate data, focussing on the placement and discovery of the original data. In this strategy, we explore the performance when data is fully replicated across all sites i.e. an entire copy of the data set is placed at each site

in the infrastructure. We assume a master-slave set-up with the master site situated in the East Coast US data centre. Unlike the other strategies, the latency experienced depends on the read/write request ratio. We provide results for a 100% read request workload, where any request can be handled successfully by any site. This provides an indication to the best possible latency that can be achieved.

Performance

The main goal of dynamic data placement and discovery is to reduce the network latency of users accessing application data. In order to evaluate Skyler, we explore its behaviour under a variety of conditions using the static workload generator. We perform a parameter exploration and evaluate the latency performance of the various strategies under different locality and offset parameters.

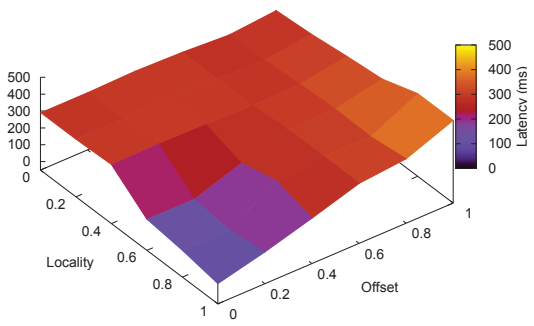
Parameter exploration. Figure 5.10 reports the 90th latency percentile for different locality (x-axis) and offset (y-axis) parameters under the five strategies. Each point in a 3D plot is obtained by running the simulator and generator with the specified parameters and averaging the results of three runs.

Figure 5.10(a) shows the plot for the geo-spatial partitioning strategy, It indicates a correlation between latency and the locality and offset parameters. As expected under high locality, the latency depends on the offset parameter. Intuitively this means that when interest in data items is concentrated to small areas of influence, the origin of these areas is important to the final performance. As the origin of interest moves away from the originally assigned location (where its site placement is based), the network path length between user and application data is increased, and thus performance deteriorates.

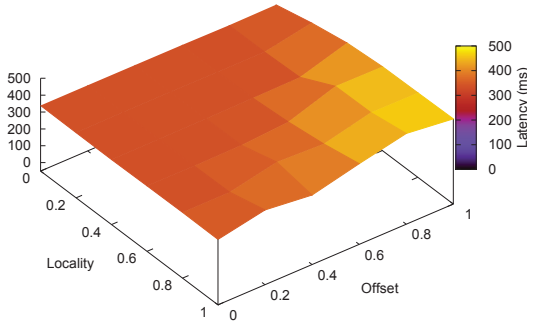
Under a low offset workload, it is the locality parameter that dictates the performance of this strategy. When users request data items that are geographically local, geo-spatial partitioning performs well. However, when the area of influence for data items increases, users request data that is placed at distant sites.

Figure 5.10(b) shows the results for the hash-based partitioning strategy, in which requests are likely to be forwarded from a user's closest site to a random site. It exhibits low performance regardless of the locality and offset parameters, with an approximate 90th latency percentile of 340 ms. However, performance drops further under a high locality and offset workload (reaching a latency of 480 ms) due to the workload performing distant requests.

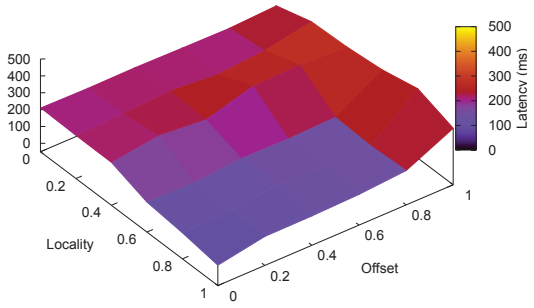
Figure 5.10(d) shows the results for Skyler with the closest site heuristic. The plot shows an improved (or similar) performance over geo-spatial and hash-based partitioning throughout the parameter space. In general, Skyler performs well regardless of the offset parameter, with locality being the driving force. Higher locality ensures that data items are more likely to have clearly defined request pattern similarities to other data items. This allows Skyler to dynamically co-locate data items on the same partition. Low locality introduces more randomness in the user-data interactions and is therefore harder for Skyler to produce



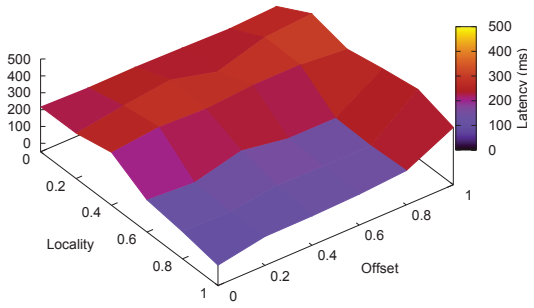
(a) Geo-spatial partitioning



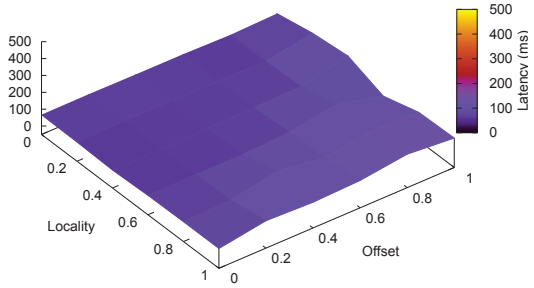
(b) Hash-based partitioning



(c) Skyler (Closest Site)



(d) Skyler (Best Partition)



(e) Full replication (100% read requests)

Figure 5.10: The 90th percentile of latency for five data placement and discovery strategies under various locality and offset parameters

high cohesion partitions. In cases where high cohesion partitions are generated from a low locality workload, partition placement is less obvious. This is because partitions are handling requests from geographical spread users.

The exception to this trend is under a high offset workload, with the 90th percentile latency reaching 260 ms regardless of the locality parameter. Further investigation shows that this is due to the variability of results under such conditions: many users request data items without a nearby site. When observing lower percentiles or the median, the latency follows the same trend as described before.

Figure 5.10(c) shows the results for Skyler with the *best partition* heuristic. The latency results match those of the previous Skyler discovery strategy, with the exception of a higher latency at $L = 0.4$. This is due to the geographical distribution of users and the increased randomness that they experience in their requests. Under these conditions, a user's best partition changes frequently and is therefore a bad heuristic for predicting data placement. The latency penalty for selecting the incorrect site when it is geographically far away from a user, is greater than if the site is geographically nearby i.e. with the closest site (CS) heuristic.

The plot shows that there is a minimum benefit in maintaining front-end application request statistics under a wide range of static workloads. As Skyler actively migrates partitions towards users, the majority of requests can be handled from a user's closest site. In rare in which this does not apply, requests are broadcast for both heuristics.

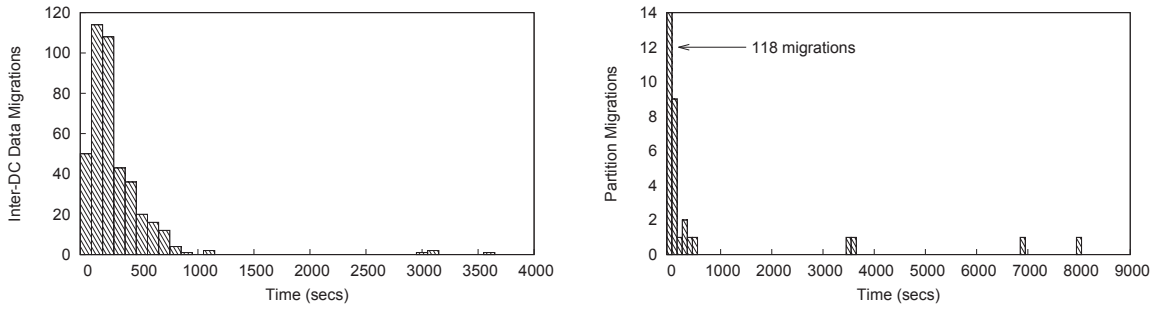
Finally Figure 5.10(e) shows the full replication strategy with a 100% read request workload. The reported latencies show stability throughout the parameter space, with the 90th percentile latency ranging from 90 to 110 ms. There is, however, minor degradation in performance with high locality and offset, which is similar to that of hash-based partitioning. This is due to the workload having longer distance requests.

These results demonstrate how Skyler can provide users with low latency application data access for a variety of workloads. Its performance is comparable to a full replication strategy with 100% read requests under a high locality, which would require six times the resources of Skyler. It is therefore likely to exceed this performance under a more realistic read/write request workload. Skyler also provides consistently lower latencies compared to hash-based and geo-spatial partitioning under all parameter configurations, demonstrating its ability to leverage patterns in a workload.

Migration cost

Dynamic data placement performed by the data partitioning and placement components of Skyler involve a cost. These components perform data migrations: the former to ensure high cohesion of partitions and the latter to reduce the network path length between users and partitions.

To evaluate the number of migrations that occur in the system, we explore a single point in



(a) Inter-DC data migrations

(b) Partition migrations

Figure 5.11: Inter-DC data and partition migrations over time under a static workload ($L = 0.8$ and $P = 0.4$)

the locality-offset parameter space ($L = 0.8$ and $P = 0.4$) further. The initial placement of data items is performed using the same technique as geo-spatial partitioning. We therefore expect data items to be re-organised due to the high offset in the workload.

Figure 5.11 shows the number of migrations that are performed by SKYLER over time. Figure 5.11(a) shows the number of inter-DC data migrations executed by the workload-aware data partitioning component. There is a steady decrease until 1000 seconds, and after that there are almost no further migrations. This is because a steady high cohesion state is identified by the distributed SKYLER instances.

Figure 5.11(b) shows the number of partition migrations that occur in the experiment. The partition placement algorithm begins by performing 118 migration. This is because of the lack of request frequencies stored in the AFVs of partitions leading to pre-mature migrations. However, after 300 seconds, an appropriate placement is identified, with only the occasional single migration thereafter.

These results demonstrate that our elastic partition and geo-aware partition placement algorithms are able to identify an ideal placement for data amongst partitions and sites. Furthermore, our two algorithms can converge under a static workload.

Scalability

Next we explore the scalability of SKYLER by evaluating the resource consumption and performance obtained when increasing the scale. We perform this exploration with three experiments that vary the number of users, data items and data centres. All scalability experiments are performed with the same experimental set-up as outlined in Section 5.5.2 and a static workload ($L = 0.8$ and $P = 0.2$). Memory consumption due to the partitioning overhead is measured using a commercial Java profiler.[10]

User scalability. Figure 5.12 shows the 95th percentile of latency (right y-axis) and total memory consumption in megabytes for maintaining all partitions (left y-axis), with an

[10]YourKit Java Profiler, `http://www.yourkit.com/`, last accessed: 20/6/2013

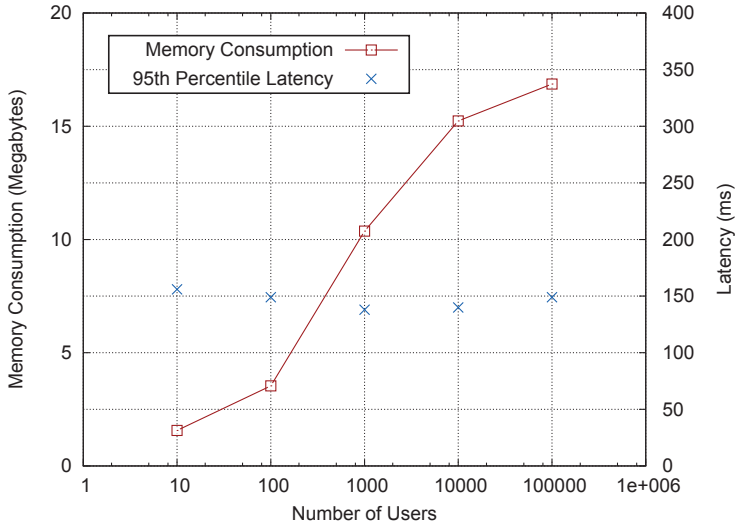


Figure 5.12: Scalability of Skyler illustrating the memory consumption overhead for all partitions and the 95th percentile latency performance with an increasing number of users

increasing number of users (log scale x-axis). In this figure, we can see that the memory overhead for maintaining DFVs and AFVs in Skyler grows linearly, trailing off as we reach $10,000+$ users. This is due to the number of requests in each DFV reaching the maximum window size and an increasing number of requests made to the same data items i.e. increased request overlap.

There are two factors that influence memory consumption for maintaining the DFVs and AFVs of partitions: the DFV window size ($window_{DFV}$) and the amount of request overlap for data items and partitions. The worst case scenario for a DFV is to store a single request frequency for each user, which results in a DFV of $window_{DFV}$ dimensions i.e. its maximum size. Conversely, the best case is for all the requests to be issued by a single user, which results in a DFV of one dimension. The worst case scenario for an AFV is: (1) for all the DFVs of data items stored in the partition to be a worst case scenario; and (2) requests from users in one DFV are not in another DFV. In this case, the number of dimensions in the AFV is the number of data items multiplied by their maximum size i.e. $window_{DFV}$ (see Section 3.3.3). However, once the number of dimensions reaches the set maximum, any further increase has no effect on memory consumption.

The figure also shows a constant 95th percentile of latency during the increase, which shows that Skyler can scale with the number of users.

Data scalability. For Skyler to be a practical solution, it must scale with the number of data items. We therefore evaluate the latency impact (right y-axis) and memory consumption overhead (left y-axis) when maintaining partitions with an increasing number of data items (log scale x-axis). With each increase in the number of data items, we increase the number of partitions to keep partition sizes constant.

Figure 5.13 shows the results: there is an approximate linear trend in the partition maintenance overhead. This is as expected, because each data item adds a fixed sized vector

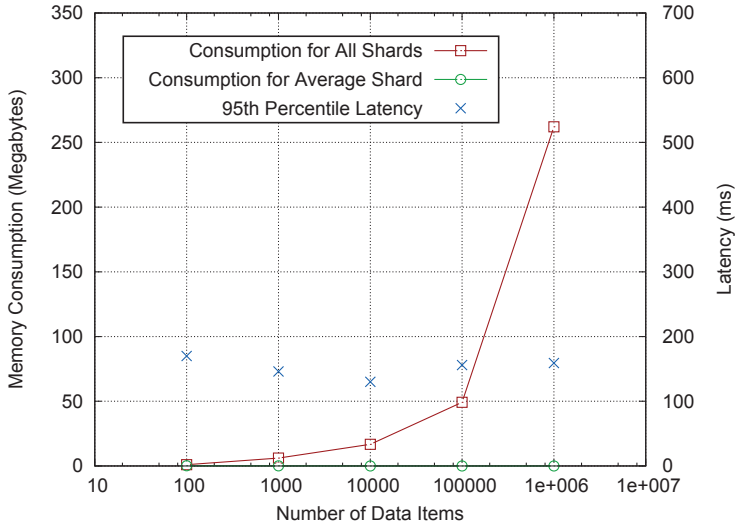


Figure 5.13: Scalability of Skyler illustrating the memory consumption overhead for all partitions, average partition and the 95th percentile latency performance with increasing data items

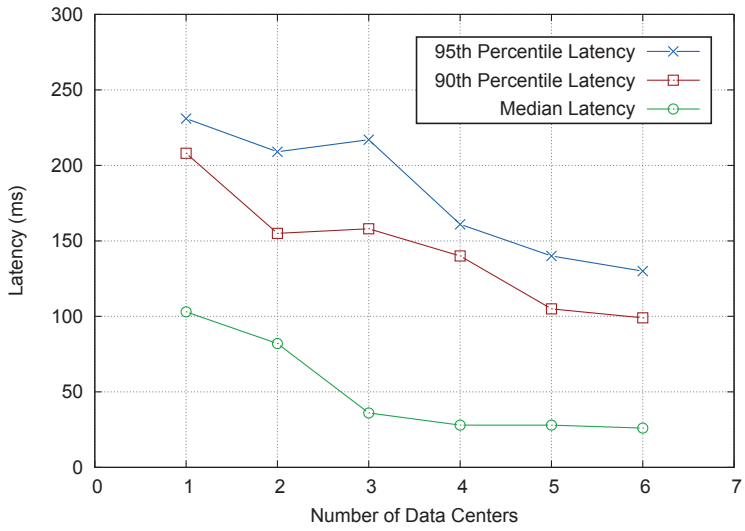


Figure 5.14: Scalability of Skyler, illustrating the median, 90th and 95th percentiles of latency with an increasing number of data centres

consisting of $window_{DFV}$ dimensions. It is important to note that we report the memory overhead when maintaining *all* partitions. In reality, however, this overhead is divided by the number of Skyler instances. The figure also shows that the 95th latency percentile remains constant with the number of data items under the same workload parameters.

Data centre scalability. The final scalability experiment evaluates the latency improvement when increasing the number of data centres. In order to perform this experiment, we assign each of the 6 data centres a ranking according to its catchment region: (1) East Coast US, (2) West Coast US, (3) Germany, (4) Japan, (5) Brazil and (6) Australia. We evaluate the added benefit when increasing the number of data centres according to their ranking.

Figure 5.14 shows the results for this experiment, providing the median, 90th and 95th

latency percentiles with each added DC. While the trend shows an improvement in performance, the amount of benefit decreases considerably after 3 and 4 DCs. This is due to the skewed distribution of users, with DCs in Japan and Brazil providing benefit to only a few users. An interesting observation is that the 90th and 95th percentile *increase* with 3 DCs. As Skyler is given a choice regarding the placement of data between the US and Europe, the performance of a minority of users is compromised for the greater good. This is shown in the large latency reduction of the median.

5.5.3 Dynamic simulation

The evaluation of Skyler under a variety of static workloads demonstrates its ability to be a *generic* data placement and discovery solution. However, online services have dynamic workloads, with large and frequent shifts occurring in user-data interactions. It is therefore important that we explore Skyler in such dynamic scenarios, evaluating its ability to *adapt* to changes in the workload.

In the analysis of global and dynamic workloads (see Section 3.1), we summarised the work of Brodersen et al. who provide insights into the geo-spatial locality shifts that occur in the YouTube service [BSW12]. More specifically, the authors report a relationship between the geographical spread in interest of videos and their popularity, describing a *spray-and-diffuse* pattern. This pattern involves (1) a video first becoming popular in a single region; (2) with traction, increases geographical spread and moves into multiple regions; and (3) loses popularity again resulting in focussed attention from a single region. This is consistent with the figures that they present, showing view entropy—a metric for describing the geographical spread or information gain—and focus—the fraction of views in a single region—with regards to the number of views (or popularity).

We therefore develop a dynamic workload generator that models such a pattern, achieved by extending the static workload generator from Section 5.5.2. We extend the static workload generator in two ways: (1) we add support to model the dynamic shifts in popularity of data items; and (2) relate popularity to both our locality (L) and offset (P) parameters.

To capture the frequent changes in data popularity, we use a variation of the ranking model [FFM06], which has been generalised to support an arbitrary ranking criterion [RFF+10]. In this model, Ratkiewicz et al. introduce *rank shifting*, in which, with a given probability ρ, data items are assigned a new ranking between their current rank and 1 (chosen uniformly). All items below the new rank are shifted down. The process repeats, allowing data items to burst in popularity by achieving a higher ranking and others to lose their popularity by repeatedly being shifted down the ranks. The author's empirical evaluation of this technique shows that it can accurately model the popularity bursts found in popular online services such as Wikipedia. We use a Zipf distribution for our ranking function with data rankings initially shuffled.

Figure 5.15 illustrates the rank shifting process with n items ordered by rank. The rank

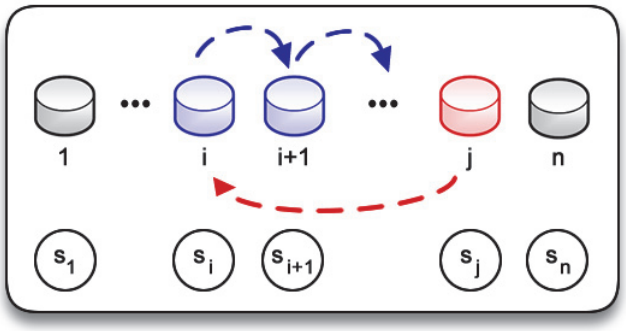


Figure 5.15: Rank shifting example with items ranked from 1 to n

ordering of items are organised with the highest on the left and lowest on the right. The figure shows item j being shifted to rank i (red dotted arrow), and thus moving all items with ranks below i down by one (blue dotted arrows) i.e. item with rank i to rank $i+1$, item with rank $i+1$ to rank $i+2$ etc. The figure also shows the probabilities of selecting items to be shifted in circles i.e. s_1 to s_n. These probabilities are assigned according to a ranking function, which in our case is a Zipf distribution. This probability differs to ρ, which is the probability for any rank shift occurring i.e. the number of rank shifts occurring within a fixed time frame.

To encompass the changes in the geo-spatial locality properties of the workload, we attach the locality and offset parameters to the rank shifting model. We achieve this by ensuring that the locality parameter follows the same Zipf distribution as popularity, and therefore depends upon a data item's ranking. Highly ranked and popular data items have low locality, with interest spread across a large geographical region. Low ranked data items have high locality, with interest focussed in a smaller geographical region.

Popularity-based locality provides a variable area of influence. To model the *movement* of interest, we must therefore turn to offset. According to the spray-and-diffuse pattern outlined by Brodersen et al. *offset shifts* occur in unison with popularity shifts. We move the centre of a data item's area of influence with each rank shift. Centres are located in the area of influence of a random node. This ensures that popular data items shift greater distances (as they have larger areas) then those that are unpopular. Figure 5.9 illustrates the offset shift of an area of influence for a single data item. Locality in our dynamic workload generator is also variable according to the data item's current popularity rank.

Experimental set-up

The experimental runs performed with the dynamic workload generator include 500 virtualised users, executing 2 million requests for $10,000$ data items, stored across 6 globally distributed data centres. All of the experiments are performed using a data frequency vector (DFV) window size of 200 ($window_{DFV} = 200$) and a site frequency vector (SFV) window size of 200 ($window_{SFV} = 200$). Data items are divided into 500 partitions of size 20, with

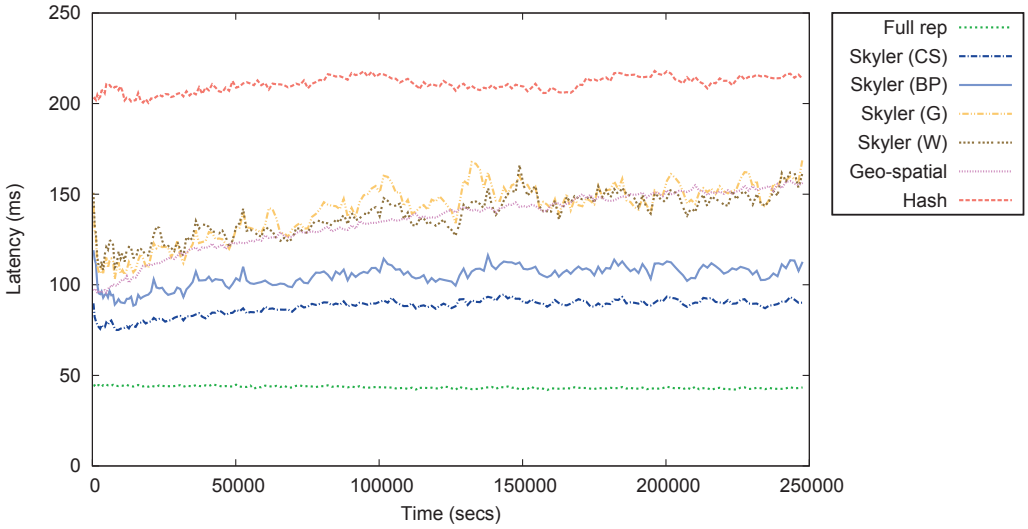


Figure 5.16: Average latency over time for seven strategies with a dynamic workload ($\rho = 10^{-4}$)

the elastic partition algorithm given a 15% expansion/reduction (see Section 5.2.2). This results in a maximum partition size of 24 and minimum partition size of 16.

Our dynamic workload generator uses a Zipf distribution with a skew of 0.8 for data popularity, and rank shifting ρ is performed with a probability of 10^{-4}, performed with every 100 requests. We also experiment with different rank shifting probabilities and their effect on latency. The workload begins with a locality parameter of 0.8 and offset of 0.2 before any rank shifting occurs. Finally, we compare these strategies with the ones from the static simulation section (see Section 5.5.2), in addition to two informed data discovery methods for Skyler: greedy and weighted DAC discovery.

Performance

Time series. To illustrate the latency with a time series, we report the results of our evaluation as the average latency of requests over time. Latencies are calculated by averaging bins of 10,000 requests. Figure 5.16 shows the results for seven different strategies with a rank shifting probability ρ of 10^{-4}. The figure depicts a constant latency for both hash-based partitioning and full replication strategies. This is expected: as we have shown, the latency for these strategies remains stable with various workload parameters (see Figure 5.10). Hash-based partitioning provides high yet stable average latencies of 200 to 220 ms, and full replication shows a stable low latency of 45 ms, both with minimal change over time.

Statically placing data across multiple sites according to the current workload can lead to an outdated placement. With each geo-spatial shift in a workload, the original placement becomes less effective. This can be seen with the geo-spatial partitioning strategy, which we model to perform perfect static placement. The figure shows that although the strategy provides low average latencies for requests on deployment, the performance slowly deteriorates over time. Over an increased length of time, its average latency reaches that of hash-based

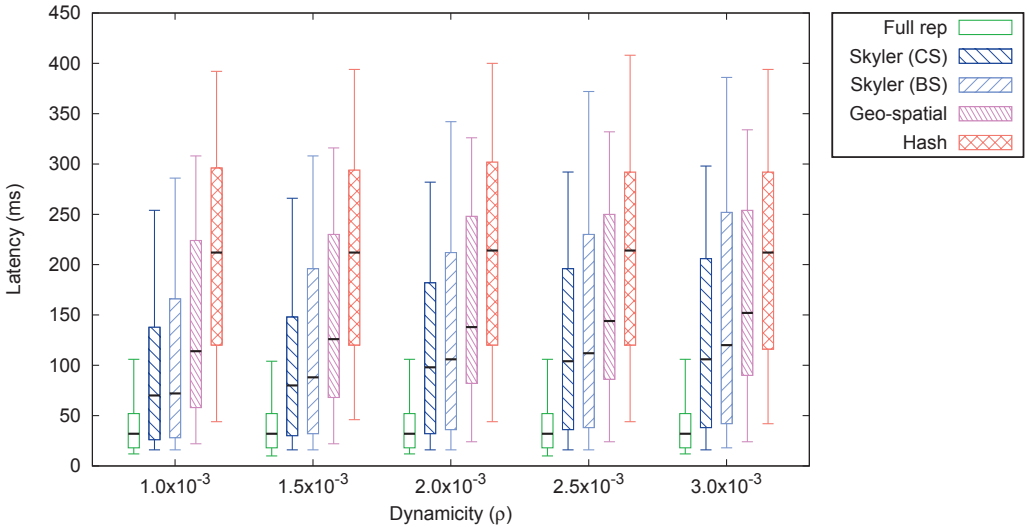


Figure 5.17: Latency box-and-whiskers plots against workload dynamicity ρ for different strategies. Boxes show 25th, median and 75th percentiles, and whiskers show 9th and 91st percentiles

partitioning. This is because the original configuration has no correlation with the latest workload.

An effective dynamic data placement strategy should be able to adapt to workload changes, providing near constant latency throughout its deployment. The figure shows the performance of four variations to Skyler: best partition (BP); closest site (CS); greedy discovery (G); and weighted discovery (W). The results show that, while two of our approaches (BP and CS) provide consistent performance, the other two (G and W) mirror the performance of geo-spatial partitioning.

The CS heuristic is the best performing variation of Skyler, providing an average improvement of 20 ms over the BP approach. This is due to its robust site selection. The BP approach exhibits lower performance due to the lack of temporal locality in the requests of individual users. As our workload generator is data-centric and has a skewed popularity workload, many requests that are issued by users are for popular data items with large geographical coverage. Users are therefore unlikely to be able to extract a best partition accurately, and are better off with having the closest site as their initial site.

This problem also applies to the greedy (G) and weighted (W) approaches, which use the best partition heuristic to select their initial site. Furthermore these approaches use a site ranking, which is based on the similarity of partitions. The results show that the constructed site rankings are unable to capture the dynamics of the workload. On closer inspection, we find that, although data items in partitions are highly correlated, the correlations between partitions are low. Approaches that base their forwarding decisions on these partition correlations are, therefore, likely to be less accurate. In choosing the incorrect initial site, the greedy (G) and weighted (W) approaches continue forwarding requests sequentially to the next best site, and so on. Ranking sites incorrectly can therefore inflict users with a high network latency penalty.

Dynamicity. We explore the latencies obtained by different data placement and discovery strategies under workloads of varying dynamicity. The dynamicity of a workload is modelled through the rank shifting probability ρ. Figure 5.17 illustrates the latency box-and-whisker plots for five data placement and discovery strategies under five workloads with varying dynamicity: 1.0×10^{-4} to 3.0×10^{-4} with increments of 0.5×10^{-4}. The range for ρ was based on a study of the Wikipedia workload [RFF+10], in which Ratkiewicz et al. show $10^{-5} \leq \rho \leq 10^{-3}$. In the figure, each box-and-whisker plot shows the 9th, 25th, median, 75th and 91st latency percentiles for a given strategy and workload. The plots show the final 50% of requests made, thus involving 1 million training requests before reporting the distribution for the remaining 1 million requests.

The figure shows that the full replication strategy providing the lowest latencies consistently across all percentiles and workloads with a median latency of 32 ms. This is what we would expect as requests can be handled by any site and are unaffected by the dynamicity in the workload. Hash-based partitioning has the highest latencies consistently across all percentiles and values for ρ with a median latency of 213 ms. The strategy is also unaffected by the workload as the placement of data across sites is random.

The geo-spatial partitioning strategy shows an increase in the median latency as the dynamicity of the workload increases. The largest increase in all the percentiles is with a dynamicity between 1.0×10^{-4} and 2.0×10^{-4}. This is because, as time passes, the dynamic workload becomes random with respect to the static placement of data. The more dynamic the workload is, the quicker this occurs. For $\rho >= 2.0 \times 10^{-4}$, the workload has already reached a random state before the training time is over.

The latency distribution for the two Skyler strategies (CS and BS) shows an increase in the median, 75th and 91st latency percentiles, with an increasing dynamicity in the workload. Both strategies have low 9th and 25th latency percentiles throughout, with 16 ms and 26-42 ms, respectively. The Skyler best partition discovery strategy (BP) has the highest latencies of the two. The 75th percentile is, however, lower than that of geo-spatial partitioning throughout. Furthermore the 91st percentile is lower than geo-spatial partitioning when $\rho < 2.0 \times 10^{-4}$. The high variance in latency is because of the lack of temporal locality in the workload, resulting in Skyler being unable to choose the best partition for users accurately. The Skyler closest site strategy (CS) shows the lowest median latency of all strategies (other than full replication). Although the 75th and 91st percentiles increase with dynamicity in the workload, they remain comparatively low. The figure also shows the closest site strategy (CS) has a lower variance in latencies compared to other strategies.

To conclude, dynamicity in workloads has no effect when adopting full replication or hash-based partitioning strategies. A geo-spatial partitioning strategy provides higher latencies under an increasingly dynamic workload but only up to a certain condition. Once the workload has become random, with respect to the static placement of data, the increase in latency also becomes minimal. The Skyler best partition discovery strategy performs badly under a highly dynamic workload due to the lack of temporal locality. Finally, the

Skyler closest site strategy provides consistently low latencies by dynamically placing data across partitions and partitions across sites. While this strategy can provide benefits for highly dynamic workloads, the largest improvement over a static strategy—such as geo-spatial partitioning—is when dynamicity is moderate i.e. $\rho <= 2.0 \times 10^{-4}$.

5.6 Discussion

The proposed algorithms, overall system and their evaluation presented in this chapter have a number of limitations. Next we discuss these limitations and provide potential solutions.

Temporal locality. All three components—i.e. workload-aware data partitioning, geo-aware partition placement and non-deterministic data discovery—and their algorithms require temporal locality in the workload. For data items to be co-located on the same partition and therefore correlated, their request patterns must be similar. However, for this correlation to have any positive effect on the latency of requests, request patterns must subsequently repeat. The same applies to our partition placement algorithm, which relies on previous requests to predict future request patterns. Although temporal locality is common in workloads (especially at a user-level), this is a limitation of our system.

Load-balancing. While we have an algorithm for dynamic data partitioning across sites, we have omitted support for load-balancing that would typically occur at a data centre. For a shared-nothing architecture to perform well, partitions must have both request locality and load-balancing properties [SC11]. A single request from a user should therefore be handled by a single partition and multiple requests from users should be distributed uniformly across partitions (and thus machines). Skyler resolves the former, however, leaves load-balancing as an open problem. A possible solution to this is to augment the partition placement algorithm to balance requests between machines. This could be achieved by leveraging the collected request frequency statistics for each partitions i.e. AFVs.

Parameter space. The proposed Skyler system provides a number of configurable parameters, which may require offline testing before deployment. For example, the elastic partition algorithm (see Section 5.2.2) requires parameters that specify the maximum and minimum number of data items that can be held in a partition. The configuration of these parameters for a given workload adds complexity to the deployment.

PlanetLab latency model. The latency model used to evaluate Skyler and the various other strategies was gathered from the PlanetLab platform. PlanetLab has known limitations [SPBP06] such as that its network links do not represent those commonly found on the Internet (see Section 4.5). Nodes and their network resources can also become overloaded, resulting in variable latency measurements between hosts.

Furthermore the PlanetLab latency model restricted our ability to conduct a large-scale evaluation of our system, due to the model consisting of only 342 nodes (see Section 5.5.1). The geographical distribution of these nodes is considerably skewed, as noted in Section 5.5.1:

most nodes densely populate a few regions. This limits the potential latency benefit that our system can provide, with fewer migrations required between sites.

Workload dataset. We have been unable to find any publicly available request log datasets that contain the IP addresses or geographic locations of users. Data protection legislation typically bars the release of such information without explicit consent from users. We require this information,however, in order to simulate accurately the latency of users and the geo-spatial properties of their interests and interactions. As a result we have resorted to developing a configurable static and dynamic workload generator for large-scale globally distributed systems.

5.7 Summary

Popular online services aim to provide globally distributed users with low latency access to application data. These services have therefore moved toward multi-DC deployments in an attempt to handle user requests from geographically close DCs and thus reduce the network path lengths between users and application data. For this to be applicable, the data required must be stored at the appropriate DC and therefore depends on the data placement strategy adopted.

Existing data placement strategies are either static (e.g. geo-spatial and hash-based partitioning), resource inefficient (e.g. full replication), ignore network latency (e.g. SPAR and SCHISM), or are offline approaches (e.g. Volley). These strategies are therefore unable to achieve the latency requirements under a global and dynamic workload, common amongst today's services.

In this chapter, we proposed an online data placement and discovery strategy that dynamically adapts its placement configuration according to the current workload. We realised this strategy with Skyler, a geo-dynamic partitioning middleware system designed to be deployed across multiple sites. The strategy entails coupling the DAC approach with a novel online and distributed elastic partition algorithm to produce workload-aware data partitions. Data items within DAC partitions share similar access patterns and thus provide high access locality.

Following this, a geo-aware partition placement policy places the dynamically constructed DAC partitions at sites according to the origin of past requests. Placement is achieved by performing a weighted Euclidean mean calculation with respect to the geographical location of sites and nearby users. The motivation for this is to reduce actively the network path lengths between users and the partitions that they access most frequently. Finally we explored various techniques to discover dynamically placed data, outlining the trade-offs between existing deterministic approaches and other non-deterministic and informed approaches.

We evaluated Skyler with a range of static workloads, demonstrating its ability to consistently provide lower latencies over other existing strategies. Furthermore we showed how

Skyler converges in the number of migrations that it performs and its ability to scale linearly with the number of users and data items.

Lastly, we evaluated Skyler with a dynamic workload, illustrating the performance over time of various existing placement strategies and Skyler with different discovery strategies. We showed that our middleware can adapt according to this workload, providing a constant low latency, whilst static strategies, such as geo-spatial partitioning, yield progressively worse latency over time.

Chapter 6

Conclusion

In this thesis, we have investigated the reduction of user-perceived latencies when accessing stored data on online services and applications. More specifically, we have focussed on services with a multi-site infrastructure in a wide-area network. We have observed that latency in such an environment depends on the data placement and discovery strategies adopted. The strategic placement and informed discovery of data across sites can shorten the network path lengths between users and data. This results in improved access latencies and an enhanced user experience for online services.

Designing effective data placement and discovery strategies is a hard open problem. Strategies must attempt to align generated configurations with the characteristics of the workloads. Our analysis of workloads has shown them to be both *complex* and *dynamic*. Existing strategies achieve static configurations, which are unable to adapt and therefore provide deteriorating performance once deployed. Other strategies are either resource inefficient or ignore the workload completely. In this thesis, we have explored the dynamic placement and discovery of data, focussing on strategies that are *workload-aware* and capable of adapting their configurations during a deployment.

We investigated the optimisation of network latencies by focussing on two application scenarios. Both scenarios serve globally distributed users and aim to provide these users with low latency access to data, retrieving either fresh or dynamic data. Although they share the same problem statement, their multi-site infrastructures and data management requirements differ.

Chapter 4 introduced the first application scenario, which explored the vision of the "Internet of Things" (IoT). This is a paradigm that calls for the worlds sensors and actuators embedded throughout our environment to collaborate and communicate amongst themselves, users and autonomous applications. A critical step towards this vision, is for the development of a platform that can *discover* fresh sources of sensor data with low latency—defined as a *global sensor discovery platform*. For this application scenario, we have focussed on the development of a data discovery strategy that organises a large number of data sources into an infrastructure that can be searched.

Global Sensor Space, our global sensor and discovery platform, interconnects the sea of sensors and data sources into a unified self-organising infrastructure. Through this platform, we have introduced workload-aware overlay networks (WAON): a method for dynamically organising nodes and their application-level links in an overlay network according to the current workload. GSS constructs its WAON using our workload-aware *data activity correlation* (DAC) approach. We have demonstrated GSS is able to adapt to a variety of workloads, providing global users with low latency discovery to fresh data source metadata.

Chapter 5 introduced the second application scenario, which explored the management of application data in popular online services such as Facebook, Twitter and Google. These services leverage the geographical locations of data centres to provide globally distributed users with low latency access to data. For this scenario, we have developed a data placement strategy that organises stored application data across a handful of highly-provisioned data centres.

Skyler, our geo-dynamic partitioning middleware, introduced a dynamic data placement strategy. It performs the active migration of application data across a multi-DC deployment to reduce access latencies. We have presented workload-aware *dynamic* partitioning, an online technique for subdividing application data into partitions according to our DAC approach. In combining this technique with an online distributed algorithm, we have shown DAC partitions remain correlated, providing high request locality throughout their deployment. In actively enhancing placement of DAC partitions across DCs, Skyler can provide globally distributed users with consistently low latency access to data, under a variety of static and dynamic workloads.

Both Chapters 4 and 5 focussed on reducing network latencies by either improving the discovery of data, as in Chapter 4, or by improving the placement of data, as in Chapter 5. An improved data discovery strategy is to forward user requests across sites (each storing statically placed data) in a manner which shortens the total network path length between users and the requested data. An improved data placement strategy is to place data across sites in order to shorten the network path lengths between users and the requested data. We have shown that both strategies have achieved their goals using the DAC approach introduced in Chapter 3. Furthermore, for these goals to be met it is important that both data placement and discovery strategies operate in unison.

Overall, this thesis has demonstrated that the optimisation of user-perceived latencies in a global multi-site infrastructure is achievable with the appropriate data placement and discovery strategies. We have shown that with dynamic workload-aware strategies, configurations can adapt under complex global and dynamic workloads.

6.1 Thesis summary

We have begun this thesis by motivating the importance of network latency for online services and applications in the context of two application scenarios: global sensor discovery platforms

and popular online services. We observe that the data access latency experienced by users in multi-site infrastructures depends on the data placement and discovery policies.

Our analysis of current workloads of online services has demonstrated that the interaction between user and data is driven by an unprecedented number of factors, which we have categorised into the characteristics of users, applications, interactions and the data. Overall, we have shown that workloads are complex, global and dynamic, and thus online services require informed dynamic placement and discovery policies.

We have presented *data activity correlation* (DAC), a workload-aware mechanism for capturing patterns in the request workloads. It provides a correlation measure that determines the relationship between data and its strength, without requiring any analysis on the data itself. Correlations are calculated by measuring the *similarity* between the access patterns of data. This makes DAC a *generic* mechanism, applicable to a wide range of application scenarios. A limitation of DAC is its need for temporal locality in the request workload to ensure that correlations can be identified.

Our experimental exploration of DAC has demonstrated its ability to identify clusters of related data in both real world and synthetic datasets. Furthermore we have shown that, in coupling DAC with a clustering algorithm, the generated clusters have near-optimal request locality. We use the features of DAC to introduce two workload-aware placement and discovery strategies, each addressing a different application scenario.

The first system presented in this thesis is Global Sensor Space, a global sensor discovery platform that interconnects the planet-wide sources of data into a unified searchable infrastructure. GSS exposes a flexible query model to globally distributed users and autonomous applications used to provide low latency discovery of data source metadata. Results returned by GSS are assured to be accurate and fresh because the platform does not cache or replicate metadata. GSS achieves these features by organising data sources into a self-organising overlay network that is workload-aware, termed a *workload-aware overlay network* (WAON). GSS utilises the DAC approach to construct a WAON, consisting of multiple hierarchical clusters of correlated data sources. We introduce a distributed and online split-and-merge algorithm to ensure high correlations within DAC cluster is maintained dynamically.

We have evaluated the GSS platform in simulation using a real latency model—obtained from PlanetLab—to emulate the features of a WAN. We demonstrate that GSS can adapt its infrastructure under workloads with both geo-spatial and semantic locality in order to provide lower data source discovery response times over a static overlay network.

The second system presented in this thesis is Skyler, a geo-dynamic partitioning middleware designed to manage the application data of popular online services. Skyler leverages the geographical placement of data centres (DC) in a multi-DC deployment to provide globally distributed users with low latency access to data. By operating on the original application data, performance benefits occur when accessing both popular and tail content. Skyler achieves these features through a placement strategy consisting of: (1) workload-aware dynamic data partitioning, generating and maintaining DAC partitions according to

our workload-aware approach, which ensures that partitions have high request locality; and (2) geo-aware partition placement, placing DAC partitions at DCs that are geographically closest to their requesting users, thus reducing the network path length between them. A limitation of dynamic placement strategies is the added complexity in discovering data. We have therefore explored various non-deterministic data discovery strategies with Skyler that reduce the network inefficiencies involved in either broadcasting requests or maintaining a global forwarding table.

We have evaluated the Skyler middleware in simulation using two workload generators and a latency model collected from PlanetLab. The static workload generator synthesises a wide range of geo-spatial workloads based on latency and distance models of geographically dispersed nodes. The dynamic workload generator synthesises geo-spatial shifts in workloads according to changes in data popularity.

We demonstrate that Skyler can provide users with consistently low latencies compared to hash-based and geo-spatial partitioning for a variety of static workloads. Under a high geo-spatial locality workload, Skyler has comparable performance to a full replication strategy with 100% read requests, regardless of the displacement of locality i.e. offset. We show that the number of data migrations performed under these static workloads converges, finding a steady state. Furthermore we show the memory overhead of maintaining DAC partitions, which increases linearly with the number of users and data items. Using the dynamic workload generator, we demonstrate Skyler's capabilities in achieving low latency with popularity and geo-spatial locality shifts in the workloads. We show how this behaviour differs from static placement policies such as geo-spatial partitioning.

6.2 Future work

With continued global growth in Internet connectivity and the deployment of increasingly interactive applications, the optimisation of user-perceived latencies will remain critical for online services and applications. Thus dynamic data placement and discovery has a place in both industry and research for the foreseeable future. We propose a number of extensions to the work presented in this thesis, in addition to addressing some of its limitations.

Data replication and placement. WAN replication of data is essential for large-scale online services and applications in order to achieve high availability. However, many services maintain more replicas than the 2- or 3-way replication typical adopted for missing critical services. Facebook, for example, performs full replication with a factor proportional to the number of DC in their infrastructure (see Section 2.2.2). The motivation for replicating data is to: improve read request scalability and ensure that globally distributed users experience low response times.

A future direction for Skyler is to investigate the minimum replication factor required to balance system-wide resource efficiency, availability and read request latency. As we have demonstrated, Skyler provides comparable latencies to that of a full replication strategy

with a 100% read request workload, without replicating data. However, determining the number and placement of replicas to achieve optimal read request performance is an open challenge. Skyler is uniquely positioned to achieve this because WAN placement of replicas for low latency access is orthogonal to the data placement problem addressed in this thesis. We can therefore leverage, for example, our geo-aware placement algorithm to determine the next best placement of partition replicas. Other challenges include the fast and efficient maintenance of replicas across a WAN, also influenced by placement.

Dynamic partition load-balancing. To ensure that a shared-nothing architecture maintains scalable, it is important that the partitioning of data across servers provides request locality and load-balancing [SC11]. Designing strategies to encompass both features is a hard problem, with dynamic workloads adding further complexity. For example, hash-based partitioning strategies such as consistency hashing (see Section 2.4.1) provide load-balancing features but lack request locality.

Skyler has thus far focussed on the placement of partitions across sites in a WAN. We propose to extend these placement capabilities for a LAN, balancing the request load across servers with dynamic placement. Partition load can be determined through the request rate of partitions, easily calculated using the statistics gathered by the Skyler middleware.

Workload-aware cache prefetching. Content distribution networks (CDNs) and distributed caches reduce the user response times of online services by placing replicas of popular content at multiple geographically dispersed "edge" servers [NSS10]. This replication process is typically performed on-demand, with frequent user requests for content resulting in the content being pushed to the relevant cache. Prefetching of content can be found on modern CDNs and distributed caching systems [Wan99, CZ03], predicting future requests based on the recent request history of individual users.

The DAC approach in this thesis (see Section 3.3) provides a measure for determining the similarity of access patterns between data items. The approach is therefore an ideal mechanism for the development of a prefetching algorithm, which decides which data items to prefetch according to their similarity to those already cached. This is an interesting use case for the DAC approach and one that we plan to explore further.

Globally distributed workload generator. The general lack of global-scale request log datasets in the public domain has made the evaluation of distributed data management systems in a wide-area network challenging. Furthermore, existing workload generators [Coo10, LLSG07, KXP12] have thus far focussed on modelling popularity shifts, various read-write ratios and the inter-arrival rates of requests. They, however, ignore the geographical distribution of users and their relationship to the data that they request. Both are major factors in determining the network path lengths between users and data, and thus critical for the evaluation in a WAN.

The workload generators described in this thesis are capable of synthesising a wide range of static and dynamic geo-spatial locality properties. Although they address many limitations of existing generators, the approach is centralised. We propose to extend the workload gener-

ators to support the orchestrated online workload generation over multiple distributed nodes. Such a generator could be deployed over a popular research platform, such as PlanetLab, emulating a large number of geographically spread users.

Publications

[BP13] Nicholas Ball, and Peter Pietzuch. Skyler: Dynamic, Workload-Aware Data Sharding across Multiple Data Centres. In *European Conference on Computer Systems (EuroSys) Poster*, Prague, Czech Republic, 2013. ACM.

Bibliography

[Aba12] Daniel J Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *IEEE, Computer*, 45(2):37–42, 2012.

[ABCD96] V. Almeida, A. Bestavros, M. Crovella, and A. De Oliveira. Characterizing Reference Locality in the WWW. In *Parallel and Distributed Information Systems (PDIS)*, pages 92–103, Miami, FL, 1996. IEEE.

[ACM04] K Aberer and P Cudré-Mauroux. Gridvine: Building Internet-Scale Semantic Overlay Networks. In *International Semantic Web Conference (ISWC)*, Hiroshima, Japan, 2004.

[ACMD+03] K Aberer, P Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: A Self-Organizing Structured P2P System. *ACM SIGMOD Record*, 32(3):29–33, 2003.

[ADJS10] Sharad Agarwal, John Dunagan, Navendu Jain, and Stefan Saroiu. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *Networked Systems Design and Implementation (NSDI)*, pages 2–2, San Jose, CA, 2010. USENIX.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Elsevier, Computer Networks*, 54(15):2787–2805, October 2010.

[ALA+11] A Ahmed, Y Low, M Aly, V Josifovski, and AJ Smola. Scalable Distributed Inference of Dynamic User Interests for Behavioral Targeting. In *Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD)*, pages 114–122, San Diego, CA, 2011. ACM.

[Ama13] Amadeus. World Travel Trends Report. Technical report, 2013.

[AMD04] Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. LAND: Stretch (1+e) Locality-Aware Networks for DHTs. In *Symposium on Discrete Algorithms (SODA)*, pages 550–559, New Orleans, LA, 2004. ACM.

[AMO13] M-dyaa Albakour, Craig Macdonald, and I Ounis. Identifying Local Events by Using Microblogs as Social Sensors. In *Open Research Areas in Information Retrieval (OAIR)*, pages 173–180, Lisbon, Portugal, 2013. ACM.

[And08] C Anderson. *The Long Tail: Why the Future of Business is Selling Less of More*. Hyperion Books, 2008.

[AP13] TG Armstrong and V Ponnekanti. LinkBench: a Database Benchmark Based on the Facebook Social Graph. In *Special Interest Group on Management of Data (SIGMOD)*, pages 1185–1196, New York, NY, 2013. ACM.

[AR04] Konstantin Andreev and Harald Räcke. Balanced Graph Partitioning. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 120–124, Barcelona, Spain, 2004. ACM.

[ASC+10] Karl Aberer, Saket Sathe, Dipanjan Chakraborty, Alcherio Martinoli, Guillermo Barrenetxea, Boi Faltings, and Lothar Thiele. OpenSense: Open Community Driven Sensing of Environment. In *Workshop on Geographic Information Systems (GIS)*, pages 39–42, San Jose, CA, November 2010. ACM.

[Ash09] K Ashton. That 'Internet of Things' Thing. *RFiD Journal*, 22:97–114, 2009.

[AT99] G Adomavicius and A Tuzhilin. User Profiling in Personalization Applications Through Rule Discovery and Validation. In *Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD)*, pages 377–381, San Diego, CA, 1999. ACM.

[AWBG07] E Adar, DS Weld, BN Bershad, and SS Gribble. Why We Search: Visualizing and Predicting User Behavior. In *International Conference on World Wide Web (WWW)*, pages 161–170, Banf, Alberta, 2007. ACM.

[AXF12] B Atikoglu, Y Xu, and E Frachtenberg. Workload Analysis of a Large-Scale Key-Value Store. In *Special Interest Group on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 53–64, London, UK, 2012. ACM.

[BBCF11] Jason Baker, Chris Bond, J Corbett, and JJ Furman. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, Asilomar, CA, 2011. ACM.

[BBPSV04] A Barrat, M Barthélemy, R Pastor-Satorras, and A Vespignani. The Architecture of Complex Weighted Networks. *National Academy of Sciences of the United States of America*, 101(11):3747–3752, March 2004.

[BC03] S Boag and D Chamberlin. XQuery 1.0: An XML Query Language. Technical report, W3C, 2003.

[BEH06] JA Burke, D Estrin, and M Hansen. Participatory Sensing. In *Workshop on World-Sensor-Web (WSW)*, pages 117–134, Boulder, Colorado, 2006. ACM.

[BF01] SR Buss and JP Fillmore. Spherical Averages and Applications to Spherical Splines and Interpolation. *ACM Transactions on Graphics (TOG)*, 20:95–126, 2001.

[BG80] PA Bernstein and N Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *International Conference on Very Large Data Bases (VLDB)*, pages 285–300, Montréal, Québec, 1980. IEEE.

[BG81] Philip a. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, June 1981.

[BGC12] Henry Blodget, Pascal-Emmanuel Gobry, and Alex Cocotas. The Future of Mobile. *Business Insider*, pages 4–5, 2012.

[BGLL08] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment*, (10), October 2008.

[BHG87] PA Bernstein, V Hadzilacos, and N Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Boston, MA, 1987.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and M Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks. In *International Conference on Weblogs and Social Media (ICWSM)*, San Jose, CA, 2009. AAAI.

[BKKN08] Lars Backstrom, Jon Kleinberg, Ravi Kumar, and Jasmine Novak. Spatial Variation in Search Engine Queries. In *International Conference on World Wide Web (WWW)*, pages 357–366, Beijing, China, 2008. ACM.

[BMEWL11] Thierry Bertin-Mahieux, Daniel PW Ellis, Brian Whitman, and Paul Lamere. The Million Song Dataset. In *International Society for Music Information Retrieval Conference (ISMIR)*, pages 591–596, Miami, Florida, 2011. University of Miami.

[BMR03] M Bawa, GS Manku, and P Raghavan. SETS: Search Enhanced by Topic Segmentation. In *Research and Development in Informaion Retrieval (SIGIR)*, Toronto, Canada, 2003. ACM.

[BP08] Nicholas Ball and Peter Pietzuch. Distributed Content Delivery Using Load-Aware Network coordinates. In *Workshop on Real Overlays & Distributed Systems (ROADS)*, volume 58, pages 1–6, Madrid, Spain, 2008. ACM.

[BPRD08] M Botts, G Percivall, C Reed, and J Davidson. OGC® Sensor Web Enablement: Overview and High Level Architecture. In *GeoSensor Networks*, pages 175–190. Springer, 2008.

[BR07] M Botts and A Robin. OpenGIS Sensor Model Language (SensorML) Implementation Specification. *OpenGIS Implementation Specification OGC*, 2007.

[Bre00] EA Brewer. Towards Robust Distributed Systems. In *Principles of Distributed Computing (PODC)*, page 7, Portland, Oregon, 2000. ACM.

[BSW12] Anders Brodersen, S Scellato, and M Wattenhofer. Youtube around the world: geographic popularity of videos. In *World Wide Web (WWW)*, pages 241–250, Lyon, France, 2012. ACM.

[BZ05] Derek J Balling and Jeremy Zawodny. *High Performance MySQL*. Safari Tech Books Online, 2005.

[Cat11] R Cattell. Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[CBC95] Carlos R Cunha, Azer Bestavros, and Mark E Crovella. Characteristics of WWW Client-Based Traces. Technical report, Boston University Computer Science Department, 1995.

[CCLS11] Zhiyuan Cheng, James Caverlee, Kyumin Lee, and Daniel Z Sui. Exploring Millions of Footprints in Location Sharing Services. In *International Conference on Weblogs and Social Media (ICWSM)*, volume 2010, pages 81–88, Barcelona, Spain, 2011. AAAI.

[CDE+12] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Operating Systems Design and Implementation (OSDI)*, pages 1–14, Hollywood, CA, 2012. USENIX.

[Cel10] Òscar Celma. The Long Tail, Long Fail, and Long Play In the Digital Music Space. In *Music Recommendation and Discovery*, page 194. Springer, 2010.

[CGC01] A Chaturvedi, PE Green, and JD Caroll. K-Modes Clustering. *Springer, Journal of Classification*, 18(1):35–55, 2001.

[CGM02] A Crespo and H Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 23–32, Vienna, Austria, 2002. IEEE.

[CGM05] A Crespo and H Garcia-Molina. Semantic Overlay Networks for P2P Systems. In *Workshop on Agents and Peer-to-Peer Computing (AP2PC)*, pages 1–13, Utrecht, Netherlands, 2005. Springer.

[CGN05] Jason Campbell, PB Gibbons, and S Nath. IrisNet: An Internet-Scale Architecture for Multimedia Sensors. In *International Conference on Multimedia (MULTIMEDIA)*, pages 81–88, Singapore, 2005. ACM.

[CJLR08] R Chaiken, B Jenkins, PÅLarson, and B Ramsey. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *VLDB Endowment*, 1(2):1265–1276, 2008.

[CJZM10] Carlo Curino, Evan Jones, Y Zhang, and Sam Madden. Schism: A Workload-Driven approach to Database Replication and Partitioning. In *International Conference on Very Large Data Bases (VLDB)*, pages 48–57, Singapore, 2010.

[CKR07] M Cha, H Kwak, and P Rodriguez. I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System. In *Internet Measurement Conference (IMC)*, pages 1–14, San Diego, CA, 2007. ACM.

[CML11] Eunjoon Cho, SA Myers, and J Leskovec. Friendship and Mobility: User Movement in Location-Based Social Networks. In *Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD)*, pages 1082–1090, San Diego, CA, 2011. ACM.

[Coo10] Brian F. Cooper. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing (SOCC)*, pages 143–154, Indianapolis, IN, 2010. ACM.

[CP97] MA Carreira-Perpinan. A Review of Dimension Reduction Techniques. Technical report, University of Sheffield, Sheffield, UK, 1997.

[CS02] E Cohen and S Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *Special Interest Group on Data Communication (SIGCOMM)*, pages 177–190, Pittsburgh, Pennsylvania, 2002. ACM.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and TW Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *In Workshop on Designing Privacy Enhancing Technologies*, pages 46–66, Berkeley, CA, 2001. Springer.

[CVKW05] David Cheng, Santosh Vempala, Ravi Kannan, and Grant Wang. A Divide-and-Merge Methodology for Clustering. In *Principles of Database Systems (PODS)*, page 196, Baltimore, Maryland, 2005. ACM.

[CZ03] Xin Chen and Xiaodong Zhang. A Popularity-based Prediction Model for Web Prefetching. *Computer*, 36(3):63–70, March 2003.

[DCKM04] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A Decentralized Network Coordinate System. In *Special Interest Group on Data*

Communication (SIGCOMM), volume 34, pages 15–26, Portland, Oregon, 2004. ACM.

[DD06] Michel-Marie Deza and Elena Deza. *Dictionary of Distances*, volume 2006. Elsevier, 2006.

[DDL+90] SC Deerwester, ST Dumais, TK Landauer, GW Furnas, and RH Landauer. Indexing by Latent Semantic Analysis. *Journal of the of the American Society for Information Science*, 41(6):391–407, 1990.

[DG00] Junyan Ding and Luis Gravano. Computing Geographical Scopes of Web Resources. In *International Conference on Very Large Data Bases (VLDB)*, pages 545–556, Cairo, Egypt, 2000. Morgan Kaufmann.

[DG08] J Dean and S Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51:107–113, 2008.

[DHJ+07] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Symposium on Operating Systems Principles (SOSP)*, pages 205–220, Stevenson, WA, 2007. ACM.

[DM01] IS Dhillon and DS Modha. Concept Decompositions for Large Sparse Text Data using Clustering. *Springer, Machine Learning*, 42:143–175, 2001.

[DNGS03] Amol Deshpande, Suman Nath, Phillip B. Gibbons, and Srinivasan Seshan. Cache-and-Query for Wide Area Sensor Databases. In *Special Interest Group on Management of Data (SIGMOD)*, pages 503–514, New York, USA, 2003. ACM.

[DNV07] C Doulkeridis, Kjetil Norvag, and Michalis Vazirgiannis. DESENT: Decentralized and Distributed Semantic Overlay Generation in P2P Networks. *IEEE Journal on Selected Areas in Communications*, 25(1):25–34, 2007.

[DS72] Peter J Denning and Stuart C Schwartz. Properties of the Working-Set Model. *Communications of the ACM*, 15(3):191–198, 1972.

[DV10] C Doulkeridis and A Vlachou. Distributed Semantic Overlay Networks. In *Handbook of Peer-to-Peer Networking*, pages 463–494. Springer, 2010.

[EFGK03] Patrick Th. Eugster, Pascal a. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, June 2003.

[EGKM04] P. T. Eugster, R. Guerraoui, A. Kermarrec, and L. Massoulié. From Epidemics to Distributed Computing. *IEEE Computer*, 37:60–67, 2004.

[EPS+98] R Engel, V Peris, D Saha, E Basturk, and R Haas. Using IP Anycast for Load Distribution and Server Location. In *Global Internet Mini-Conference (GlobeCom)*, Sydney, Australia, 1998. IEEE.

[Fac12] Facebook Inc. Facebook Annual Report. Technical report, 2012.

[FB74] RA Finkel and JL Bentley. Quad Trees a Data Structure for Retrieval on Composite Keys. *Springer, Acta Informatica*, 4:1–9, 1974.

[FFM06] Santo Fortunato, Alessandro Flammini, and Filippo Menczer. Scale-Free Network Growth by Ranking. *Physical Review Letters*, 96(21):218701, May 2006.

[Fie00] RT Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[Fit04] B Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, (124):72–76, 2004.

[FM03] Michael J Freedman and David Mazi. Sloppy Hashing and Self-Organizing Clusters. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 45–55, Berkeley, CA, 2003. Springer.

[Fod02] Imola K Fodor. A Survey of Dimension Reduction Techniques. Technical Report 1, Lawrence Livermore National Laboratory, UCRL-ID-148494, 2002.

[For65] Edward W Forgy. Cluster Analysis of Multivariate Data: Efficiency Versus Interpretability of Classifications. *Biometrics*, 21:768–769, 1965.

[FPRU90] U Feige, D Peleg, P Raghavan, and E Upfal. Randomized Broadcast in Networks. *Random Structures & Algorithms*, 1(4):447–460, 1990.

[FSB04] GHL Fletcher, HA Sheth, and K Börner. Unstructured Peer-to-Peer Networks: Topological Properties and Search Performance. In *Workshop on Agents and Peer-to-Peer Computing (AP2PC)*, pages 14–27, New York, USA, 2004. Springer.

[FVFB05] Michael J. Freedman, Mythili Vutukuru, Nick Feamster, and Hari Balakrishnan. Geographic Locality of IP Prefixes. In *Conference on Internet Measurement (IMC)*, pages 13–13, New York, USA, 2005. ACM.

[FW99] K Finkenzeller and R Waddington. *RFID Handbook: Radio-Frequency Identification Fundamentals and Applications*. Wiley, New York, USA, 1999.

[Gar05] Jesse James Garrett. Ajax: A New Approach to Web Applications, 2005.

[GDQ92] S Ghandeharizadeh, DJ DeWitt, and W Qureshi. A Performance Analysis of Alternative Multi-Attribute Declustering Strategies. In *Special Interest Group on Management of Data (SIGMOD)*, pages 29–38, San Diego, CA, 1992. ACM.

[GMDW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, second edi edition, 2009.

[GMP+08] Jeremy Ginsberg, Matthew H Mohebbi, Rajan S Patel, Lynnette Brammer, Mark S Smolinski, and Larry Brilliant. Detecting Influenza Epidemics using Search Engine Query Data. *Nature*, 457(7232):1012–1014, 2008.

[GNOT92] David Goldberg, David Nichols, BM Oki, and D Terry. Using Collaborative Filtering to Weave an Information Tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[Gra78] JN Gray. Notes on Data base Operating Systems. In *Operating Systems*, pages 393–481. Springer Berlin Heidelberg, 1978.

[GS08] M Gjoka and M Sirivianos. Poking Facebook: Characterization of OSN Applications. In *Workshop on Online Social Networks (WOSN)*, pages 31–36, Seattle, WA, 2008. ACM.

[GTC+07] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. Does Internet Media Traffic Really Follow Zipf-like Distribution? In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, volume 35, pages 359–360, San Diego, CA, June 2007. ACM.

[Hav07] S Havens. OpenGIS transducer markup language (TML) implementation specification. *OpenGIS Implementation Specification OGC*, 2007.

[HB09] M Hefeeda and M Bagheri. Forest Fire Modeling and Early Detection using Wireless Sensor Networks. *Ad Hoc & Sensor Wireless Networks*, 7:169–224, 2009.

[HFKB12] Kurt Hornik, I Feinerer, Martin Kober, and Christian Buchta. Spherical K-Means Clustering. *Journal of Statistical Software*, 50(10):1–22, 2012.

[HHL03] Ryan Huebsch, JM Hellerstein, and N Lanham. Querying the Internet with PIER. In *International Conference on Very Large Data Bases (VLDB)*, volume 29, pages 321–332, Berlin, Germany, 2003.

[HKJR10] Patrick Hunt, Mahadev Konar, FP Junqueira, and Benjamin Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Annual Technical Conference (ATC)*, volume 8, pages 11–11, Boston, MA, 2010. USENIX.

[HKMP96] J Hromkovič, R Klasing, B Monien, and R Peine. Dissemination of Information in Interconnection Networks (Broadcasting & Gossiping). In *Combinatorial Network Theory*, pages 125–212. Springer, 1996.

[HLY06] KYK Hui, J Lui, and DKY Yau. Small-World Overlay P2P Networks: Construction, Management and Handling of Dynamic Flash Crowds. *Computer Networks*, 15:2727–2746, 2006.

[Hot33] H Hotelling. Analysis of a Complex of Statistical Variables into Principal Components. *Warwick & York, Journal of Educational Psychology*, 24(6):417, 1933.

[HRW08] B Huberman, D Romero, and Fang Wu. Social Networks that Matter: Twitter Under the Microscope. *First Monday*, 14(1), 2008.

[Hua98] Z Huang. Extensions to the K-means Algorithm for Clustering Large Data sets with Categorical Values. *Data Mining and Knowledge Discovery (Springer)*, 2(3):283–304, 1998.

[JD88] AK Jain and RC Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[JHVB11] M Jacomy, S Heymann, T Venturini, and M Bastian. ForceAtlas2, A Graph Layout Algorithm for Handy Network Visualization. *http://www.medialab.sciences-po.fr/*, 2011.

[JMF99] AK Jain, MN Murty, and PJ Flynn. Data Clustering: A Review. *ACM Computing Surveys (CSUR)*, 31(3):264–323, 1999.

[JW04] I Jawhar and J Wu. A Two-Level Random Walk Search Protocol for Peer-to-Peer Networks. In *World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI)*, pages 1–5, Orlando, USA, 2004. IIIS.

[Kar47] K Karhunen. *Über Lineare Methoden in der Wahrscheinlichkeitsrechnung*. University of Helsinki, 1947.

[Kar01] George Karypis. Evaluation of Item-Based Top-N Recommendation Algorithms. In *Conference on Information and Knowledge Management (CIKM)*, pages 1–13, Atlanta, Georgia, 2001. ACM.

[KGZY02] V Kalogeraki, D Gunopulos, and D Zeinalipour-Yazti. A Local Search Mechanism for Peer-to-Peer Networks. In *Conference on Information and Knowledge Management (CIKM)*, pages 300–307, McLean, VA, 2002. ACM.

[Kin10] J Kincaid. EdgeRank: The Secret Sauce that Makes Facebook's News Feed Tick. *TechCrunch*, April 2010.

[KK98] G Karypis and V Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[KKNG12] Juhi Kulshrestha, F Kooti, A Nikravesh, and PK Gummadi. Geographic Dissection of the Twitter Network. In *International Conference on Weblogd and Social Media (ICWSM)*, Dublin, Ireland, 2012. AAAI.

[KLL97] D Karger, E Lehman, and T Leighton. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide

Web. In *Symposium on Theory of Computing (STOC)*, pages 654–663, El Paso, Texas, 1997. ACM.

[KLPM10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, A Social Network or a News Media? In *International Conference on World Wide Web (WWW)*, pages 591–600, Raleigh, NC, 2010. ACM.

[KM02] T Klingberg and R Manfredi. Gnutella 0.6. 2002.

[Knu73] DE Knuth. *The Art of Computer Programming. Sorting and Searching, Vol. III.* Addison-Wesley, Reading, MA, 1973.

[Kor09] Y Koren. The Bellkor Solution to the Netflix Grand Prize. *Netflix Prize Documentation*, 2009.

[KPL+09] Sebastian Kaune, Konstantin Pussep, Christof Leng, Aleksandra Kovacevic, Gareth Tyson, and Ralf Steinmetz. Modelling the Internet Delay Space Based on Geographical Locations. In *Parallel, Distributed and Network-based Processing (PDP)*, pages 301–310, Weimar, Germany, 2009. IEEE.

[KSV+12] A Kaltenbrunner, S Scellato, Yana Volkovich, David Laniado, Dave Currie, Erik J Jutemar, and Cecilia Mascolo. Far From the Eyes, Close on the Web: Impact of Geographic Distance on Online Social Interactions. In *Workshop on Online Social Networks (WOSN)*, pages 19–24, Helsinki, Finland, 2012. ACM.

[KXP12] Konstantinos V. Katsaros, George Xylomenos, and George C. Polyzos. GlobeTraff: A Traffic Workload Generator for the Performance Evaluation of Future Internet Architectures. In *New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, May 2012.

[LAW02] C Lu, GA Alvarez, and J Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *File and Storage Technologies (FAST)*, Monterey, CA, 2002. USENIX.

[LC13] Raul Landa and RG Clegg. Measuring the Relationships between Internet Geography and RTT. In *International Conference Computer Communications and Networks (ICCCN)*, number February 2012, pages 1–7, Nassau, Bahamas, 2013. IEEE.

[LCC+02] Q Lv, P Cao, E Cohen, K Li, and S Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *International Conference on Supercomputing (ICS)*, pages 84–95, New York, USA, 2002. ACM.

[LCP05] EK Lua, J Crowcroft, and M Pias. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[LGBS09] Jaime Lloret, Miguel Garcia, Diana Bri, and Sandra Sendra. A Wireless Sensor Network Deployment for Rural and Forest Fire Detection and Verification. *Sensors*, 9(11):8722–8747, January 2009.

[LGW03] A Leon-Garcia and I Widjaja. *Communication Networks*. McGraw-Hill, 2 edition, 2003.

[LH07] Jure Leskovec and Eric Horvitz. Planetary-Scale Views on an Instant-Messaging Network. Technical report, Microsoft Research, 2007.

[LLSG07] Huajing Li, WC Lee, A Sivasubramaniam, and L Giles. SearchGen: A Synthetic Workload Generator for Scientific Literature Digital Libraries and Search Engines. In *Joint Conference on Digital Libraries (JCDL)*, pages 137–146, Vancouver, Canada, 2007. ACM.

[LM10] Avinash Lakshman and P Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[Loi48] M Loire. Fonctions Aleatoires de Second Ordre. *CR Acad. Sci. Paris*, 220:380, 1948.

[LSL12] Joonseok Lee, Mingxuan Sun, and G Lebanon. A Comparative Study of Collaborative Filtering Algorithms. *CoRR*, abs/1205.3:1–27, 2012.

[LSY03] G Linden, B Smith, and J York. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.

[LSZ04] A Loser, K Schubert, and F Zimmer. Taxonomy-based Routing Overlays in P2P Networks. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 407–412, Coimbra, Portugal, 2004. IEEE.

[LTN07] S Lightstone, T Teorey, and T Nadeau. *Physical Database Design*. Morgan Kaufman, 2007.

[Lug12] Edwin Lughofer. A Dynamic Split-and-Merge Approach for Evolving Cluster Models. *Springer, Evolving Systems*, 3(3):135–151, February 2012.

[LWQ10] WB Lober, Stephen Wagner, and Christina Quiles. Development and Implementation of a Loosely Coupled, Multi-Site, Networked and Replicated Electronic Medical Record in Haiti. *ACM SIGOPS Operating Systems Review*, 43(4):79–83, 2010.

[Mac67] J Macqueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, California, US, 1967.

[Mad97] T Madell. *Disk and File Management Tasks on HP-UX*. Prentice-Hall, 1997.

[MBMEL12] Brian McFee, Thierry Bertin-Mahieux, Daniel P.W. Ellis, and Gert R.G. Lanckriet. The Million Song Dataset Challenge. In *International Conference Companion on World Wide Web (WWW)*, pages 909–916, Lyon, France, 2012. ACM.

[MD88] P Mockapetris and KJ Dunlap. Development of the Domain Name System. *ACM Computer Communication Review*, 18(4):123–133, 1988.

[MFJWM04] David Malan, Thaddeus Fulford-Jones, Matt Welsh, and Steve Moulton. Codeblue: An Ad hoc Sensor Network Infrastructure for Emergency Medical Care. In *International Workshop on Wearable and Implantable Body Sensor Networks (BSN)*, volume 5, London, UK, 2004. IEEE.

[MJ09] KV Mardia and PE Jupp. *Directional statistics*. Wiley, 2009.

[MMG+07] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Internet Measurement Conference (IMC)*, page 29, San Diego, CA, 2007. ACM.

[MMG+08] John McCulloch, Paul McCarthy, Siddeswara Mayura Guru, Wei Peng, Daniel Hugo, and Andrew Terhorst. Wireless Sensor Network Deployment for Water use Efficiency in Irrigation. In *Workshop on Real-world Wireless Sensor Networks (REALWSN)*, pages 46–50, Glasgow, UK, 2008. ACM.

[MMR+08] Rohan Narayana Murty, Geoffrey Mainland, Ian Rose, Atanu Roy Chowdhury, Abhimanyu Gosain, Josh Bers, and Matt Welsh. CitySense: An Urban-Scale Wireless Sensor Network and Testbed. In *Technologies for Homeland Security (HST)*, pages 583–588, Waltham, MA, May 2008. IEEE.

[MRS08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schutze. *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008.

[MSPC12] D Miorandi, S Sicari, F De Pellegrini, and I Chlamtac. Internet of Things: Vision, Applications and Research Challenges. *Elsevier Ad Hoc Networks*, 10:1497–1516, 2012.

[New06] Mark EJ Newman. Modularity and Community Structure in Networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, June 2006.

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, and Paul Saab. Scaling Memcache at Facebook. In *Networked Systems Design and Implementation (NSDI)*, Lombard, IL, 2013. USENIX.

[NLZ07] Suman Nath, Jie Liu, and Feng Zhao. SensorMap for Wide-Area Sensor Webs. *IEEE Computer*, 40(7):90–93, July 2007.

[NSMP11] Anastasios Noulas, Salvatore Scellato, Cecilia Mascolo, and Massimiliano Pontil. An Empirical Study of Geographic User Activity Patterns in Foursquare. In *International Conference on Weblogs and Social Media (ICWSM)*, pages 70–73, Barcelona, Spain, 2011. AAAI.

[NSS10] Erik Nygren, RK Sitaraman, and Jennifer Sun. The Akamai Network: A Platform for High-Performance Internet Applications. *ACM Operating Systems Review (SIGOPS)*, 44(3):2–19, 2010.

[NZ01] T. S. Eugene Ng and Hui Zhang. Towards Global Network Positioning. In *Internet Measurement Workshop (IMW)*, pages 25–29, New York, USA, 2001. ACM.

[NZ02] TSE Ng and H Zhang. Predicting Internet Network Distance with Coordinates-based Approaches. In *Computer and Communications Societies (INFOCOM)*, New York, USA, 2002. IEEE.

[OA04] David Oppenheimer and Jeannie Albrecht. Distributed Resource Discovery on PlanetLab with SWORD. In *Workshop on Real, Large Distributed Systems (WORLDS)*, number 1, San Francisco, CA, 2004. USENIX.

[Och57] A Ochiai. Zoogeographic Studies on the Soleoid Fishes found in Japan and its Neighbouring Regions. *Bull. Jpn. Soc. Sci. Fish*, 22(9):526–530, 1957.

[OV11] M Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 2011.

[PABB05] J Porter, P Arzberger, HW Braun, and P Bryant. Wireless Sensor Networks for Ecology. *BioScience*, 55(7):561–572, 2005.

[PB07] AMK Pathan and R Buyya. A Taxonomy and Survey of Content Delivery Networks. Technical report, University of Melbourne, Melbourne, 2007.

[PCW+03] M Pias, J Crowcroft, S Wilbur, T Harris, and S Bhatti. Lighthouses for Scalable Distributed Location. In *Peer-to-Peer Systems II*, pages 278–291. Springer, 2003.

[PES+10] Josep M Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The Little Engine(s) That Could: Scaling Online Social Networks. In *Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 375–386, New Delhi, India, 2010. ACM.

[PF01] Sanjoy Paul and Zongming Fei. Distributed Caching with Centralized Control. *Computer Communications*, 24(2):256–268, 2001.

[PLM08] W Penzo, S Lodi, and F Mandreoli. Semantic Peer, Here Are the Neighbours You Want! In *Conference on Extending Database Technology (EDBT)*, pages 26–37, Nantes, France, 2008. ACM.

[PLMS06] P. Pietzuch, J. Ledlie, M. Mitzenmacher, and M. Seltzer. Network-Aware Overlays with Network Coordinates. In *International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 12–12, Lisboa, Portugal, 2006. IEEE.

[Pol07] J Polak. Mobile Environmental Sensor Systems Across a Grid Environment–the MESSAGE project. *ERCIM News*, 2007.

[Pri08] D Pritchett. BASE: An Acid Alternative. *ACM Queue*, 6:48–55, 2008.

[PSL80] M Pease, R Shostak, and L Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[RB07] O Riva and C Borcea. The Urbanet Revolution: Sensor Power to the People! *IEEE Pervasive Computing*, 6(2):41–49, 2007.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-scale Peer-to-Peer Systems. In *International Conference on Distributed Systems Platforms (Middleware)*, number November 2001, pages 329–350, Heidelberg, Germany, 2001. ACM.

[Ree78] DP Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, MIT, 1978.

[RFF+10] Jacob Ratkiewicz, Santo Fortunato, Alessandro Flammini, Filippo Menczer, and Alessandro Vespignani. Characterizing and Modeling the Dynamics of Online Popularity. *Physical Review Letters*, 105(15):158701, October 2010.

[RFH01] Sylvia Ratnasamy, Paul Francis, and Mark Handley. A Scalable Content-Addressable Network. In *Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, San Diega, CA, 2001. ACM.

[RG00] R Ramakrishnan and J Gehrke. *Database Management Systems.* Osborne/McGraw-Hill, third edit edition, 2000.

[RIS94] Paul Resnick, N Iacovou, and M Suchak. GroupLens: an Open Architecture for Collaborative Filtering of Netnews. In *Computer Supported Cooperative Work (CSCW)*, pages 175–186, Chapel Hill, NC, 1994. ACM.

[RMP07] Ian Rose, Rohan Murty, and PR Pietzuch. Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds. In *Networked Systems Design & Implementation (NSDI)*, Cambridge, MA, 2007. USENIX.

[RP08] P Raftopoulou and EGM Petrakis. iCluster: A Self-organizing Overlay Network for P2P Information Retrieval. In *European Conference on IR Research (ECIR)*, pages 65–76, Glasgow, UK, 2008. Springer.

[RT04] M Ronstrom and L Thalmann. MySQL Cluster Architecture Overview. Technical report, MySQL Technical White Paper, 2004.

[SC11] M Stonebraker and R Cattell. 10 Rules for Scalable Performance in "Simple Operation" Datastores. *Communications of the ACM*, 54(6):72–80, 2011.

[SF12] PJ Sadalage and M Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[SGD02] S Saroiu, KP Gummadi, and RJ Dunn. An Analysis of Internet Content Delivery Systems. *Operating Systems Review (SIGOPS)*, 36:315–327, 2002.

[SGM00] Alexander Strehl, Joydeep Ghosh, and Raymond Mooney. Impact of Similarity Measures on Web-page Clustering. In *Workshop on Artificial Intelligence for Web Search (AAAI)*, pages 58–64, Austin, Texas, 2000. AAAI.

[Sha48] CE Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal, Bell Systems*, 27:379–423, 1948.

[Sin84] Sinnott, R. W. Virtues of the Haversine. *Sky and Telescope*, 68(2), 1984.

[SKKR01] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-Based Collaborative Filtering Recommendation Algorithms. In *International Conference on World Wide Web (WWW)*, pages 285–295, Hong Kong, China, 2001. ACM.

[SM10] Salvatore Scellato and Cecilia Mascolo. Distance Matters: Geo-social Metrics for Online Social Networks. In *Workshop on Online Social Networks*, Boston, MA, 2010. USENIX.

[Smi82] AJ Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[SMK01] Ion Stoica, Robert Morris, and David Karger. Chord: A scalable peer-to-peer lookup service for internet applications. In *Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, San Diego, CA, 2001. ACM.

[SNLM11] Salvatore Scellato, A Noulas, Renaud Lambiotte, and Cecilia Mascolo. Socio-Spatial Properties of Online Location-Based Social Networks. In *International Conference on Weblogs and Social Media (ICWSM)*, pages 329–336, Barcelona, Spain, 2011. AAAI.

[SP11] David Smiley and DE Pugh. *Apache Solr 3 Enterprise Search Server*. Packt Publishing, 2011.

[SPBP06] N Spring, L Peterson, A Bavier, and V Pai. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. *ACM Operating Systems Review (SIGOPS)*, 40:17–24, 2006.

[SPK02] Lakshminarayanan Subramanian, Venkata N Padmanabhan, and Randy H Katz. Geographic Properties of Internet Routing. In *Annual Technical Conference (ATEC)*, pages 243–259, Monterey, CA, 2002. USENIX.

[SPL+04] Jeff Shneidman, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, Margo Seltzer, and Matt Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical report, Harvard, Technical Report TR-21-04, 2004.

[STA01] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *Computer and Communications Societies (INFOCOM)*, volume 3, pages 1801–1810, Anchorage, Alaska, 2001. IEEE.

[Sto86] M Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering*, 9:4–9, 1986.

[SWZ07] Thomas C. Schmidt, Matthias Wahlisch, and Yin Zhang. On the Correlation of Geographic and Network Proximity at Internet Edges and its Implications for Mobile Unicast and Multicast Routing. In *International Conference on Networking (ICN)*, Sainte-Luce, Martinique, 2007. IEEE.

[TCJM12] AL Tatarowicz, C Curino, Evans Jones, and Sam Madden. Lookup Tables: Fine-grained Partitioning for Distributed Databases. In *International Conference on Data Engineering (ICDE)*, pages 102–113, Washington, D.C., 2012. IEEE.

[TR03] D Tsoumakos and N Roussopoulos. Adaptive probabilistic search for peer-to-peer networks. In *International Conference on Peer-to-Peer Computing (P2P)*, pages 102–109, Linkoping, Sweden, 2003. IEEE.

[TSK07] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Pearson Education India, 2007.

[TXD03] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 175–186, New York, USA, 2003. ACM.

[VMCG09] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the Evolution of user interaction in Facebook. In *Workshop on Online Social Networks (WOSN)*, New York, USA, 2009. ACM.

[Vog09] W Vogels. Eventually Consistent. *Communications of the ACM*, 52:40–44, 2009.

[Wan99] Jia Wang. A Survey of Web Caching Schemes for the Internet. *ACM, Computer Communication Review*, 29(5):36–46, 1999.

[WBS+09] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. User Interactions in Social Networks and their implications. In *European Conference on Computer Systems (EuroSys)*, page 205, New York, USA, 2009. ACM.

[WD06] P Ward and G Dafoulas. *Database Management Systems*. Cengage Learning EMEA, 2006.

[WHMW11] S Wu, JM Hofman, WA Mason, and DJ Watts. Who Says What to Whom on Twitter. In *International conference on World Wide Web (WWW)*, pages 705–714, Hyderabad, India, 2011. ACM.

[WK83] E Wong and RH Katz. Distributing a Database for Parallelism. *ACM SIGMOD Record*, 4, 1983.

[WSPZ12] Christo Wilson, Alessandra Sala, Krishna P. N. Puttaswamy, and Ben Y. Zhao. Beyond Social Graphs: User Interactions in Online Social Networks and their Implications. *ACM Transactions on the Web (TWEB)*, 6(4):1–31, November 2012.

[XRZ+13] R Xin, J Rosen, M Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In *International Conference on Management of Data (SIGMOD)*, pages 13–24, New York, USA, 2013. ACM.

[YGM02] B Yang and H Garcia-Molina. Improving search in peer-to-peer networks. In *International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, 2002. IEEE.

[YL11] Jaewon Yang and J Leskovec. Patterns of temporal variation in online media. In *Web Search and Data Mining (WSDM)*, pages 177–186, Hong Kong, 2011. ACM.

[YMZD04] Fang Yuan, Zeng-Hui Meng, Hong-Xia Zhang, and Chun-Ru Dong. A New Algorithm to get the Initial Centroids. In *International Conference on Machine Learning and Cybernetics (ICMLC)*, pages 1191–1193, Shanghai, China, 2004. IEEE.

[YW10] S Ye and SF Wu. Measuring Message Propagation and Social Influence on Twitter.com. In *Social Informatics (SocInfo)*, Laxenburg, Austria, 2010. Springer-Verlag.

[ZHS+04] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[ZSS08] Ming Zhong, Kai Shen, and Joel Seiferas. The Convergence-Guaranteed Random Walk and Its Applications in Peer-to-Peer Networks. *IEEE Transactions on Computers*, 57(5):619–633, 2008.

[ZSW+12] X Zhao, Alessandra Sala, Christo Wilson, Xiao Wang, Sabrina Gaito, Haitao Zheng, and Ben Y. Zhao. Multi-scale Dynamics in a Massive Online Social Network. In *Internet Measurement Conference (IMC)*, pages 171–184, Boston, USA, 2012. ACM.