Dynamic Object Partitioning and Replication for Cooperative Cache

Omar Asad

Doctor of Philosophy

School of Computer Science
McGill University
Montreal, Quebec, Canada

January 2021

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

©Omar Asad, 2021

Abstract

Data intensive applications are usually designed using a multi-tier architecture that comprises a web tier, an application tier and a backend database tier. To process requests, application servers fetch data from the backend database. For faster processing, the accessed data or objects are cached at the application servers which is known as application caching. To fully utilize the aggregated cache space of all application server caches, they can connect together to build a *cooperative cache*. In such a case, an application server can read an object from its local cache or from a remote cache.

While a cooperative cache can reduce frequent access to the backend database, local cache access is still preferable over remote access because it is much faster. However, achieving a high rate of local cache access is a challenging task. Current enterprise workloads are complex and dynamic; user requests typically access several objects, different requests access overlapping object sets, object access is highly skewed and object popularity changes over time.

In this thesis, we propose mechanisms to increase the efficiency of cooperative cache solutions. In a first step, we analyze a real workload to understand the major characteristics of its type of requests and how they access objects. Furthermore, we develop a suite of request and object partitioning approaches that partition requests across servers and objects across caches to achieve a high local cache hit as well as balance application loads.

We further augment the data partition techniques with a novel data replication technique that replicates objects into caches of the application servers that need them the most. While doing so, we carefully consider different kinds of overhead such as the need to update all replicas, the overhead of different consistency protocols and the fact that cache space is limited.

Both are purely distributed as well as our replication solutions support dynamic workloads where access pattern can change; they monitor the workload and trigger a replication and/or re-replication if need arises.

We integrated our partitioning and replication solutions into an existing, cooperated cache framework, and extended the framework significantly to facilitate dynamic configuration changes.

We evaluated the various partitioning and replication approaches using the YCSB and RUBiS benchmarks, showing that our approaches are applicable in different dynamic workload scenarios and are able to autonomously capture workload changes and adapt instantly. Furthermore, experimental results show that our replication solution outperforms a well-known replication solution.

Abrégé

Les applications à grand volume de données sont habituellement conçues au moyen d'une architecture multiniveau qui comprend un niveau Web, un niveau applications et un niveau base de données principale. Pour traiter les requêtes, les serveurs d'applications récupèrent des données de la base de données principale. Afind'accélérer le traitement, les données ou objets consultés sont antémémorisés dans les serveurs d'applications; ce qui est connu comme la mise en cache. Dans le but d'utiliser tout l'espace des caches cumulés dans tous les caches du serveur d'applications, les caches peuvent être regroupés pour créer un cache coopératif. Le cas échant, un serveur d'applications peut lire un objet de son cache local ou d'un cache distant.

Même si un cache coopératif peut réduire des accès fréquents à la base de données principale, il est préférable d'accéder à un cache local qu'à un cache distant, car c'est beaucoup plus rapide. Toutefois, l'atteinte d'un fort taux d'accès à des caches locaux représente une tâche ardue. Les charges de travail actuelles sont complexes et dynamiques. Les requêtes des utilisateurs accèdent typiquement à plusieurs objets; différentes requêtes accèdent à des ensembles d'objets qui se chevauchent; l'accès aux objets est très inégal; la popularité des objets change avec le temps.

Dans cette thèse, nous proposons des mécanismes permettant d'augmenter l'efficience des solutions de caches coopératifs. Tout d'abord, nous analysons une véritable charge de travail afin de comprendre les principales caractéristiques de son type de requêtes et comment ces dernières accèdent à des objets. Ensuite, nous mettons au point une série

de méthodes de partitionnement des requêtes et des objets, qui partitionnent les requêtes de tous les serveurs et les objets de l'ensemble des caches dans le but d'obtenir un taux de réussite local élevé et d'équilibrer les charges des applications.

Puis, nous augmentons les techniques de partitionnement des données à l'aide d'une technique novatrice de réplication des données qui reproduit des objets dans des caches des serveurs d'applications qui en ont le plus besoin. Ce faisant, nous examinons soigneusement différents types de temps système, comme la nécessité de mettre à jour toutes les répliques, le temps système de divers protocoles de cohérence ainsi que le fait que l'espace de caches est restreint.

Les deux techniques sont entièrement distribuées et nos solutions de réplication conviennent aux charges de travail dynamiques où les modes d'accès changent. Elles surveillent la charge de travail et déclenchent une réplication et/ou une nouvelle réplication, au besoin.

Nous avons intégré nos solutions de partitionnement et de réplication dans une infrastructure existante de caches coopératifs, et nous avons grandement étendu l'infrastructure afin de faciliter des changements dynamiques de configuration.

Nous avons évalué les diverses méthodes de partitionnement et de réplication à l'aide des tests de performance YCSB et RUBiS, qui ont démontré que nos méthodes peuvent être appliquées à différents scénarios de charges de travail dynamiques et qu'elles peuvent capter de manière autonome des changements de charge de travail et s'adapter instantanément. De plus, les résultats des expériences démontrent que notre solution de réplication surpasse une solution de réplication bien connue.

Acknowledgements

Foremost, I would like to thank my supervisor, Professor Bettina Kemme, for her continued guidance and support throughout my PhD study. The amount of personal time she puts towards my work was very valuable. Her ability to look at the research from different angels was very helpful. Her contribution and guidance is priceless.

I extend my gratitude to my PhD committee, Professor Jörg Kienzle and Professor Derek Ruths, for their useful feedback and research ideas. They were open for discussions with no restrictions.

I'm grateful to the System Administrator at the School of Computer Science. They facilitated my access to all computing resources to run experiments. In addition, I would like to thank my colleagues at the Distributed Information System lab for their collaboration and insightful discussions.

Lastly, I would like to express my appreciation to my family, especially my mother, Afaf, and my brothers and sisters for their support and encouragement. I'm also thankful for my wife, Asma, who enormously supported me throughout my PhD journey. Finally, I'm very grateful to my two kids, Bushra and Zuhair, whose presence empowered me to finish this thesis.

Contents

1 Introduction **1**

1.1 Challenges 3

1.2 Thesis Contribution 5

1.3 Thesis Organization 8

1.4 Publications and Contributions of Students 8

2 Adaptive Cooperative Cache Framework **9**

2.1 Multitier Architecture 9

2.1.1 Request Characteristics 10

2.1.2 Caching 10

2.2 Cooperative Cache 12

2.2.1 Handling Write Requests 13

2.3 Data Structures for Cache Directory 14

2.3.1 Overview Data Structures 14

2.3.2 Choosing false positive rates for global and local bloom filters . 18

2.3.3 Comparison using a global Bloom filter vs. not using a global Bloom Filter 19

2.3.4 Experimental results 21
2.4 AdaptCache: Adaptive Cooperative Cache Framework 25
2.4.1 Request- and Object Policies 26
2.4.2 Logging 27
2.4.3 Policy Deployment 27
2.4.4 Lazy Data Migration 28
2.4.5 Workload Meta-data 29

3 Adaptive Object Partitioning and Migration **31**
3.1 Workload Analysis 32
3.1.1 Request Characteristics 32
3.1.2 Object Characteristics 35
3.1.3 Further Analysis 36
3.2 Policy Generation 37
3.2.1 Parameter Collection 39
3.2.2 Object Distribution First 41
3.2.3 Request Distribution First 44
3.2.4 Assigning Partitions to Servers 47
3.2.5 Reducing Workload Meta-data 48
3.3 Evaluation 49
3.3.1 Benchmarks 49
3.3.2 Understanding Partitioning Behavior 50
3.3.3 Algorithm Comparison 52
3.3.4 Assigning Partitions to Servers 55

3.3.5 Meta-data Pruning 57
3.3.6 Log Window Size Analysis 60
3.3.7 Result Highlights 61

4 Consistency and Space Aware Cache Replication 63

4.1 Replication Challenges 65
4.1.1 Data Consistency 65
4.1.2 Limited Cache Space 66
4.1.3 Dynamic Workload 67
4.2 AdaptCache Replication Extension 68
4.2.1 Extending the Parameter Collection 68
4.2.2 Extracting Operation Costs 70
4.2.3 Independence from Distributed Solution 71
4.3 Basic Replication 72
4.4 Managing Update Overhead 74
4.4.1 Calculating Execution Costs 74
4.4.2 Random Write Distribution 76
4.5 Managing Limited Cache Size 79
4.5.1 No Write Operations in The System 80
4.5.2 Write Operations in The System. 81
4.6 Evaluation 82
4.6.1 Experiments 83
4.6.2 Limited Cache Space Analysis 85
4.6.3 Workload Changes Triggering Partitioning 86

4.6.4 Read/Write Changes Triggering Replication 88
4.6.5 Solution Overhead 88

5 Related Work **91**

5.1 Data Partitioning and Migration in Distributed Database Systems 92
5.1.1 Data Partitioning 92
5.1.2 Data Migration 95
5.2 Data Replication 99
5.2.1 Full Data Replication 99
5.2.2 Partial Data Replication 102
5.2.3 Data Replication in the Cloud 104
5.2.4 Data Consistency for Replicated Data 104
5.3 Data and Space Management for Distributed Caches 107
5.3.1 Cache Architectures and Data Partitioning 107
5.3.2 Data Replication and Caching 110
5.3.3 Managing Cache Space 111
5.4 Dynamic System Configuration 112

6 Conclusions **114**

7 Future Work **117**

7.1 Cooperative Cache vs Stand-alone Cache 118
7.2 Optimizing Log Messages 119
7.3 Autoscaling Cache Nodes 119
7.4 Fault Tolerance 120

7.5 Applying Caching Solutions to NUMA 121

Bibliography **123**

Acronyms **135**

List of Figures

1.1 Various Access Types Latencies . . . 3

2.1 Multitier Cache Architecture . . . 10
2.2 Cooperative Cache . . . 12
2.3 Optimized Bloom Filter Directory Performance . . . 24
2.4 Adaptive Cooperative Cache Framework . . . 26

3.1 Accumulative request drift over time . . . 34
3.2 Item popularity . . . 36
3.3 Temporal Locality . . . 37
3.4 Assignining partitions to servers example . . . 48
3.5 Req-GP algorithm performance . . . 51
3.6 Partitioning Strategies . . . 53
3.7 Smart Assignment of Partitions to Servers . . . 55
3.8 Graph Construction and Partitioning Time . . . 57
3.9 RUBiS and Req-GP: pruning meta-information . . . 58
3.10 RUBiS and Obj-GP: pruning meta information . . . 59

3.11 Impact of time window size 61

4.1 Impact of number of replicas on update latency 66

4.2 Impact of limited cache space on latency 66

4.3 Partitioning vs Basic replication performance 73

4.4 75% write workload 84

4.5 75% cache space 85

4.6 Dynamic reconfiguration 87

4.7 Processing times 89

List of Tables

2.1 Global and local bloom filter configurations for $p_{all} = 10^{-3}$, $N = 4$ and various n 23

2.2 Request and object logged information 29

3.1 Collected parameters for request and object partitioning 40

4.1 Parameters for object replication 69

4.2 Cost (Latency) 71

1

Introduction

With the unprecedented increase of web users and the abundance of web applications, performance is still considered as the key factor for both user satisfaction as well as service provider success. Performance, which can be expressed by the response time experienced by the user, has crucial impact on applications [27, 25, 72, 7, 71] where spending extra milliseconds on users' requests can lead to a considerable revenue loss for the respective web application. Google, for instance, announced that a half second delay decreases their traffic by 20%[1]. Similarly, Amazon announced that every 100 ms of additional latency costs them 1% revenue loss[2]. These drops in traffic and revenue can easily translate to millions of dollars in losses[3]. The impact of performance on user behaviour was further demonstrated by a study conducted by Yahoo! Labs [3]. In summary, the lower user response times are, the more satisfied the users will be, and as a result, the more profitable the web business will be.

Many web businesses rely on a multi-tier architecture to build and manage their services [121]. In such an architecture, client requests are handled by a sequence of service components, which often include a web-service that handles web pages, an application service that executes the business logic [77], and a database system that stores business critical information. In principle, each service can have a set of actual servers and a

[1]http://glinden.blogspot.ca/2006/11/marissa-mayer-at-web-20.html
[2]https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales
[3]https://www.soasta.com/wp-content/uploads/2014/10/eBook-eCommerce-Best-Practices.pdf

load-balancing [95, 99] mechanism distributes requests across servers. In reality, such service replication is most typical for the application tier while the database tier usually remains a monolithic block, because few commercial database systems offer advanced distribution or replication support at a reasonable price.

Therefore, an important scalability issue with such architecture is that with increasing user demand, the database can become the bottleneck and harm performance. A well-established solution to tackle this scalability issue is application caching where each application server is augmented with a local cache layer that stores the data most recently fetched from the database. A subsequent request for data that is locally cached avoids expensive database access, alleviating database load and reducing user latency. However, in most current implementations each application server cache is oblivious of other server caches, and data can be replicated across caches. Thus, aggregated cache space is not well utilized and, at the same time, data inconsistency between caches and the back-end database can occur.

To overcome these limitations, a cooperative cache architecture that allows an application cache to access other application caches was developed [2]. In this architecture, each application cache maintains a subset of the objects and each application server can access objects residing in the local cache, as well as the remote caches. Only if an object is in no cache, does it need to be retrieved from the database. The main advantage of such architecture is that the full cache space is utilized.

However, data access latency can be quite different between local cache, remote cache, and database. To illustrate different latencies between various types of data access, Figure 1.1 shows the latencies in the prototype used in our research for fetching an object from a local cache, a remote cache, and a database[4]. The figure shows that both local and remote cache accesses have lower latencies than the database access. However, latency of local object access is 10 times lower than remote cache access. This is due to the fact that accessing a remote cache requires marshalling and unmarshalling of request and response messages as well as network delays. We believe that remote access

[4]using JBoss as an application server, and PostgreSQL as a database system. More details of our system are presented in Chapters 2 and 3.

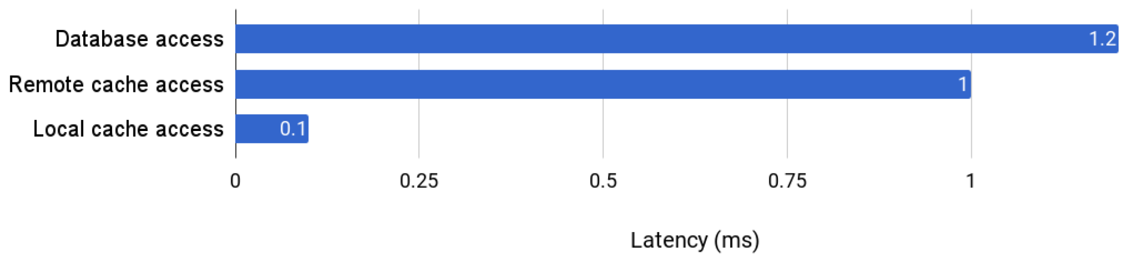


Figure 1.1: Various Access Types Latencies

has slightly less latency compared to database access because database access requires complex query processing while remote access only need to fetch the already formatted object from the cache storage. Therefore, in the ideal case, requests find most of the data they need in the local cache, find most of the other data in the remaining caches, and have to access the database as seldomly as possible. However, given the complex workloads of web businesses, achieving this ideal case is a challenging task, as we explore below.

1.1 Challenges

Exploiting the fast local cache access requires addressing the following challenges.

Distribute requests and objects across servers The first challenge is to be able to assign requests to a server and allocate objects to its cache so that requests can find as many objects as possible in their local cache. We call this locality-aware request and object partitioning. At the same time, the load across servers should be balanced. The challenge stems from the complexity of current enterprise workloads where a request accesses many objects and an object might be accessed by many different request types. To illustrate, assume there is a user searching on eBay[5] for HP laptops while another looks for new laptops. If the database contains a new HP laptop, the data sets accessed by the two requests overlap. This workload aspect, which we refer to as *request overlap*, makes the partitioning hard.

[5]https://www.ebay.com/

Object replication A well-known technique to overcome the partitioning challenge is data replication, where objects are replicated to caches that require them. Replicating objects across caches enhances the performance by allowing requests to access objects locally instead of performing costly remote or database access. However, as mentioned before, having objects in several caches induces two major challenges. The first is guaranteeing object consistency across caches in case of update requests [46]. An update request to an object located at a certain cache needs to be propagated to object replicas at other caches or needs to invalidate the replicas in other caches. If this is not done, users might be served outdated data. This step of updating or invalidating object replicas degrades the performance since an update request has to spend extra time communicating with other nodes [1]. However, the overhead varies based on the used consistency protocol. Strong consistency requires more time updating replicated objects, while a weak consistency protocol provides typically fast response to the users, but inconsistencies are allowed to occur. Thus, the challenge here is not only to take update requests into consideration when replicating an object, but also to consider the consistency requirements of the application. The second challenge for conducting object replication is that replicating objects consumes the limited cache space, which leads to a smaller number of different objects to be cached across caches, leading again to more database access. In summary, object replication, although it potentially can improve performance, needs to deal with the two major issues of data consistency and a limited cache space.

Dynamicity of Workloads Application workloads can change quickly [120] reducing the effectiveness of how data is currently partitioned and/or replicated. A workload change occurs when the relative frequency of request types changes (e.g. browsing vs purchasing) or when the data set accessed by a particular request changes. As an example of the second case, assume again that a user is searching on eBay for HP laptops with the result sorted from newest to oldest listing, showing a total of 10 laptops as the first output page. This means that newly listed items appear first. If a new laptop is inserted into the database and then the same request is submitted again, the result set will be slightly different. Another workload change aspect is a read/write request ratio change. Given such workload changes, finding a good distribution/replication assignment for re-

quests and objects is a dynamic task and placements need to be adjusted on a regular basis.

1.2 Thesis Contribution

This thesis provides novel approaches for conducting adaptive request and object partitioning as well as replica assignment in a cooperative cache environment with dynamic workloads. These approaches are fully independent of the underlying application. That is, they can work with any application, and with different number of application servers or different network setups. Furthermore, the provided solution is adaptive to workload changes, as it transparently monitors the workload and triggers the appropriate actions whenever changes have been detected. More concretely, the thesis provides the following contributions.

Workload Analysis In order to provide a practical data partitioning and replication solution, we analyze workload patterns of a popular e-commerce site, namely e-Bay[6]. The goal of conducting such analysis is to help us understand the main characteristics of requests and the objects they access as typically found in online applications. To this end, we develop a workload crawling tool that concurrently calls various eBay search APIs using specific parameters and then extract and organize the output for each API to generate useful workload statistics. We determined that search requests tend to change the set of objects they access over time. However, we found that the degree of change varies among request types as they have different search semantics. For objects, we determined that their overall popularity as well as their popularity over time vary accordingly. The adaptive request and object partitioning algorithm that we describe in this thesis take these observations into account. Moreover, we customize our benchmarks and workloads to emulate the patterns we observed.

Requests and Objects Partitioning We developed a suite of request and object partitioning solutions which aim in distributing a set of requests and their respective objects

[6] https://www.ebay.com/

across a set of application servers such that most requests can usually find most of their data in the local caches, and load is equally distributed across these servers. We leverage both graph and hyper-graph data structures [26, 91] to represent the relationships between requests and the objects they access. We explore two solution spaces. The first one is request distribution first, by which we let nodes in the graph represent requests in the workload and edges represent objects. The idea is to place requests that are accessing the same objects close to each other in the graph/hyper-graph. Then we use a graph/hyper-graph partitioning library to partition the graph into sub-graphs, each of which contains a subset of the workload requests that are accessing overlapping sets of objects. We further develop a greedy algorithm to assign the respective objects across these partitions according to access frequency. The second solution space that we explore is object distribution first. That is, we build a graph/hyper-graph by mapping each object in the workload to a vertex in the graph/hyper-graph and then let the requests represent the edge/hyper-edges. Thus, objects that are accessed by the same request will be close to each other in the graph. Then, similar to what we have done in the first solution space, we use a graph/hyper-graph library to partition the object graph/hyper-graph into sub-graphs, each of which contains a set of objects that are closely related. We again use a greedy algorithm to distribute the respective requests across object partitions in a way that maximizes the local cache access. The two solution spaces aim at minimizing the number of remote cache accesses while balancing the load across partitions.

Object Replication Since perfect data partitioning, that would completely prevent remote cache access, is impossible [26, 87, 44, 61], we develop a novel object replication solution that is orthogonal to the used request and object partitioning technique. Our replication solution assumes that the underlying requests and objects are reasonably well distributed across caches. From there, we decide about replicating a set of objects that are accessed by requests assigned to different partitions. The proposed solution decides for each of these objects, to which partitions, if any, it should be replicated to. To do so, we evaluate the trade-off between the gain of replicating an object and the performance overhead for maintaining data consistency. Based on this evaluation, the solution replicates objects so that the overall response time is kept low. The solution is oblivious to

the deployed consistency protocol. As such, it adapts to various consistency setups. In addition, in case of limited cache space, the solution takes into account the trade-off of adding an object replica to a full cache, which reduces access for this object but requires evicting another object that now must be retrieved from the database whenever needed.

Adaptive Caching Framework We extend an existing cooperative cache (CC) framework [2] to support a fully dynamic and adaptive caching solution. In order to do so, we develop a workload tracking mechanism that continuously observes which requests are executed and what objects they access, and detects workload changes. We categorize workload changes into either a general workload change or a read/write change. A general workload change occurs when the request type distribution or object popularity have changed for some objects, leading to a decreased cache hit ratio. Thus, re-partitioning of requests and objects is triggered to improve the cache hit ratio. As re-partitioning may lead to a costly object migration from one cache to another, we assign new partitions to caches in a way that keeps the number of objects that need to be moved low. We also reduce the processing time of the partitioning step by ignoring requests that do not lend themselves to caching because, for example, they are not popular or they change their result set too frequently. A read/write change, on the other hand, occurs when the cache hit ratio does not change but the ratio of objects reads compared to object writes changes. In this case, we do not repartition but only reconsider which objects to replicate.

Implementation and Evaluation We have implemented our solution and integrated it into the CC framework, now called AdaptCache. The original CC uses several open source components: JBoss as an application server, Apache web-server[7] as front-end load-balancer and Ehcache[8] as a caching component. In order to add adaptability, we extended Ehcache to support migration of objects and Apache web-server to support dynamic request distribution.

To show that our solutions are practical, we have conducted extensive experiments

[7] https://httpd.apache.org/
[8] http://www.ehcache.org/

using two popular benchmarks; RUBiS[9] and YCSB[10]. Both were adapted to better emulate dynamic workloads. The experimental results show that our partitioning and replication solutions greatly enhance performance in terms of user perceived latency as well as resource utilization compared to existing solutions. In addition, a comparison between different partitioning solutions shows that, in general, most partitioning solutions behave reasonably well. However, performance can vary depending on several factors, such as the complexity of the workload, the frequency of workload changes, and the type of workload change.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 presents the overall architecture of Adapt-Cache, the adaptive caching framework we use, and while doing so introduces some necessary background information. Chapter 3 presents the workload analysis part as well as a suite of partitioning solutions and compares their performance in detail. Chapter 4 presents the consistency and space-aware replication solution and its performance results. Chapter 5 presents the related work. Chapter 6 concludes the thesis and finally, Chapter 7 highlights several directions for future work.

1.4 Publications and Contributions of Students

I am the sole student contributor of all the research presented in this thesis. The contributions in Chapter 3 have been published in [5]. I am the sole student author on this paper. Bettina Kemme was supervising the work and supported me in writing the paper.

[9] rubis.ow2.org

[10] https://github.com/brianfrankcooper/YCSB

2

Adaptive Cooperative Cache Framework

This chapter follows a step-wise approach to present the adaptive caching framework this thesis is utilizing and enhancing. We first present a typical multi-tier architecture cache, then show how this architecture is extended to construct a Cooperative Cache (*CC*) framework. From there, we show how CC is augmented with an external adaptive monitoring and controlling tool that is capable of hosting data partitioning and replication solutions, monitoring workload, and triggering an appropriate action once a workload change is detected.

2.1 Multitier Architecture

Multitier architectures are widely used as a typical building paradigm for enterprise applications [121]. They facilitate managing different parts of the application as well as scaling these parts horizontally and vertically. Figure 2.1 depicts the standard architecture model. In this model, user requests, typically issued through web-browser or programming APIs, are first intercepted by a front-end load-balancer (*LB*). The LB distributes these requests across a set of application servers (*AS*) that are horizontally scaled. Each of these *AS* contains the same business code that can handle the different requests and returns the result to the user through the LB. Request execution often requires communicating with the backend database (DB) that maintains durable application data.

2.1.1 Request Characteristics

Generally, a request represents a user-initiated action which is executed by the business logic and returns a result to the client. In a web application, a request is usually submitted over http, indicating a URL. This URL string typically includes the domain name, the targeted page (method), and parameter names along with their values. Thus, we can say that a request calls a specific business method with certain values for its input parameters. In this thesis, we say that a particular request initiated by a client is called a “request instance”, and all request instances that have the same URL belong to the same *request type*. During execution within the AS, each request instance is tagged with a unique UIUD. If it is clear from the context, we use "request" and "request type" interchangeable.

2.1.2 Caching

Without caching, the AS has to communicate with the DB every time a request requires access to data. This DB access is a costly operation that can negatively affect user response time. In addition, as the load increases, more and more requests are concurrently accessing the *DB* which causes a bottleneck that further worsens the response time. This is where caching can have a potential benefit on the overall system performance, and as a result on user response time.

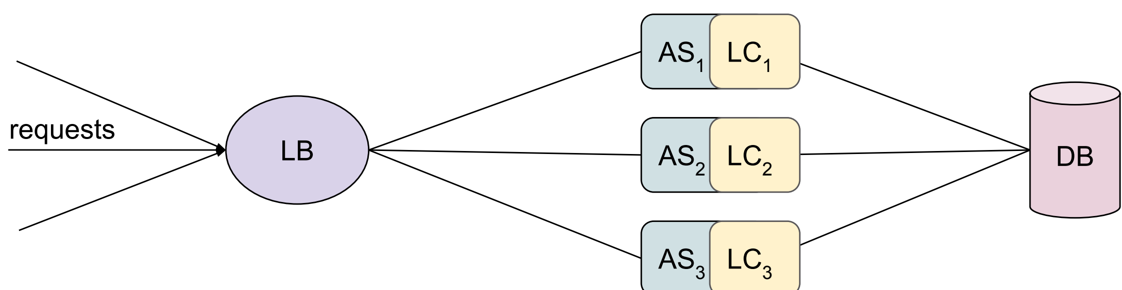


Figure 2.1: Multitier Cache Architecture

One widely used caching architecture is application caching which augments each

AS with a local cache (*LC*) instance as depicted in Figure 2.1. The local cache keeps the objects that were accessed by recently executed requests. User requests typically trigger database access through SQL queries which can be quite complex; this means that the AS does not usually know immediately which objects a request will access. Thus, a standard mechanism is to execute the query on the database but to request only the primary keys of the objects instead of retrieving the entire objects. With the identifiers determined, the AS gets the actual objects from the local cache, and those that are not in the cache are retrieved from the database through a direct primary key look-up. At the same time, these objects are loaded to the cache to speed up future queries. Thus, the main advantage of this architecture is that it can avoid object transfer from the DB as long as the objects are stored locally.

However, using independent local caches causes each server to load objects individually, not being aware of the cache content of other servers nor being able to access their caches. If the number of objects accessed by the requests assigned to the server is bigger than the cache capacity of its local cache, it will come to thrashing. The LC will fill up quickly, triggering cache eviction of objects that might be accessed again shortly after. Such evict and reload leads to database access that can be avoided were the cache big enough. With this, user response times become longer and the database more loaded. Could a server also access the cache of another server, this load on the database could be reduced.

Another effect of independent local caches is that objects will be replicated across the servers. This is particularly the case if the LB dispatches user requests across ASs using a content blind distribution mechanism, such as round robin, that is not aware of what each cache stores and what objects are accessed by a request. Then requests at different ASs could load the same object into their caches, as caches are not aware of each others content. With this, a considerable number of objects might be replicated.

However, replication can lead to inconsistencies in case of write requests. If one AS receives a write request, it will only update its local copy and the database, while other caches replicas remain unmodified as the cache is oblivious of the contents of other caches. This can lead to stale caches [32] and harm the correctness of the application.

To overcome these limitations, [2] extended the basic multitier caching architecture by letting different LCs communicate with each other in order to form a Cooperative Cache, or *CC*.

2.2 Cooperative Cache

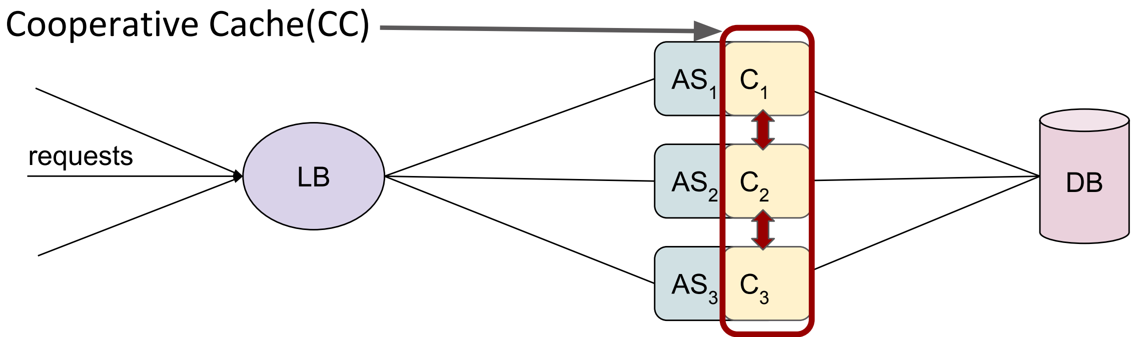


Figure 2.2: Cooperative Cache

The Cooperative Cache, shown in Figure 2.2, is a multitier distributed caching architecture that allows all AS caches to communicate with each other in order to have a cluster of cache nodes. CC allows any cache to manipulate objects stored in any cache in the cluster. When an AS requests an object that resides in one of the caches, it is retrieved from that cache. Only if none of the caches has it, it will be loaded from the database and then stored in the local cache of that AS.

One important issue within CC is how each cache can keep track of objects loaded to or deleted from other caches. For this purpose, CC maintains a cache directory at each cache that keeps track of the cache content of all caches. Whenever an object is loaded to a cache, the cache multicasts a message to all other caches, which adds the object identifier along with its location to their cache directories. Similarly, once an object is evicted from a cache, the cache multicasts a message about this eviction to all caches which remove the entry for this object from their directories. When the AS requests an object from the local cache, the local cache will first check, using the cache directory, if

the object is stored locally. If this is the case, the object will be directly returned from the local cache. If the object is not present in the local cache but is in a remote cache, the local cache will fetch the object from that cache and then return it to the AS without storing it locally. If the object is neither present at the local cache nor at the remote caches, a cache fault occurs. The AS will load it from the database. It will also store it in the local cache.

2.2.1 Handling Write Requests

Having a remote cache access capability, the cooperative cache needs to handle write requests and guarantee a consistent state between cached objects and their respective durable copies in the database. To this end, we extend the read-write cache strategy as introduced by Hibernate[1] to handle write requests at any cache. Whenever a cache receives a write request and before performing the actual write on the database, it replaces the value of the object, located either in a local or a remote cache, with a soft lock. In the mean time, if another request tries to read or write to this locked object, the request has to go directly to the database to perform the respective operation. Note that we rely on the database management system to guarantee an appropriate isolation level for concurrent transactions. Once the write has been performed at the database, the value of the locked object in the cache is replaced with the newly updated object and other requests can resume their work on this updated object. As a result, the cached object becomes consistent with its respective database copy.

Overall, this is a strong consistency policy. Clearly, other mechanisms are possible, that may be less or more costly. Our system is independent of the actual write-consistency mechanism in place, and finding a good update mechanism was not part of this thesis.

Should an object be cached at several locations, additional mechanisms have to ensure that different caches do not have different values. We describe in more details replica consistency mechanism in Chapter 4.

[1] http://www.ehcache.org/documentation/2.7/integrations/hibernate.html

2.3 Data Structures for Cache Directory

The cache directory is replicated at each cache and is responsible of keeping track of all objects across caches as well as their respective locations. Here, we will discuss three data structures that can be used as cache directory and compare their time and space performances.

2.3.1 Overview Data Structures

HashMap The original version of Cooperative Cache [2] uses a HashMap that stores key/value pairs. The key is the object identifier and the value is the set of object locations. Hence, an object that is not cached does not have en entry, an object that is cached at a single location, i.e. not replicated, will have an entry with a single location value and an object that is replicated will have an entry with a multiple locations' value.

Adding an object key along with its value is done with a simple put operation. Internally, Hashmap in Java, which is the language used in the cache layer of our system, uses an array of buckets to store entries [79]. When adding an entry, the put operation calculates first the hashcode of the entry key and based on the generated hashcode, it finds a bucket and stores the entry into that bucket. Note that HashMap stores both the key and its respective value into the bucket.

Retrieving an object location from a hashmap is done through a get operation. The get operation calculates first the hashcode of the object's key to find the respective bucket that should ideally store the entry. If the entry exists in that bucket, it returns a single object location in case of a non replicated object or multiple locations in case of a replicated object. Deleting an object from the hashmap is conceptually similar. The delete operation calculates the hashcode of the key and then searches for the respective bucket for that entry and if exists, it removes it.

An advantage of using hashamp is the time efficiency for adding and retrieving objects as these operations only require calculating a hashcode of an object's key and then

retrieve its actual location. However, a downside of utilizing hashmap as a cache directory is the amount of memory space required as each object is represented as a key/value pair. This space consumption, in turn, decreases the overall memory space allowed for caching objects themselves. That is, the more space is used for the cache directory, the less space is available for the actual objects.

Bloom Filter To overcome the space consumption issue, we look in this thesis at bloom filters [14] for the cache directory. Bloom filters have been proposed for caching in the past [40]. Bloom filter is a fast and space-efficient data structure that allows easy look-up for keys within a set. A bloom filter is a bit array of m bits. When an element (in our case object), is added to a set (in our case a cache), the element key is used as input to k hash-functions, each returning an integer smaller than m, setting the corresponding array positions of the bloom filter to 1.

To check whether an element is a member of the set, its key is again input to the k hash functions. If the corresponding array positions returned by these hash-functions are all set to one, then the probability is high that the element is member of the set. If any of these positions returns false, then the element is certainly not a member of the set. There can be false positives but not false negatives.

The probability of a false positive can be limited by choosing appropriate values of m and k, given the number of elements in the set. In particular, given a desired (maximum) false positive rate p, and an expected number n of objects to be stored, the optimal number of bit m can be calculated as[2]:

$$m = \frac{n * ln(p)}{(ln2)^2} \tag{2.1}$$

that is, it is proportional to the number of elements in the set. And the optimal number of hash functions k can be calculated as:

$$k = m/n * ln(2) \tag{2.2}$$

[2] https://en.wikipedia.org/wiki/Bloom_filter

or

$$k = -\frac{ln(p)}{ln2} = -\log_2(p)$$

The original bloom filter can not perform deletion by simply resetting a bit to zero again because the bit could have been set by multiple elements. As in our caching system deletion can occur, we need to use a generalized version of bloom filter known as counting bloom filter [41]. It extends the array positions (m) from being a single bit to multi-bit counters. To add an element, the corresponding counters are increased by one. To check an element existence, if all of the corresponding counters are non-zero, there is a high probability that the element is stored. Otherwise, it is certainly not stored. To delete an element, the corresponding counters are decremented by one. To prevent a counter overflow, the size of the counter can be configured to a number of bits that should be large enough to handle the max number of elements that can set a certain counter. Thus, a counting bloom filter requires an order of magnitude more space compared to the original bloom filter.

In a distributed scenario with c different local caches, each storing a subset of the objects, past work [40] created the cache directory by having a bloom filter for each of the caches. In our approach we also create these filters, referring to them as *local* bloom filters. Additionally, we create what we call a *global* bloom filter which keeps track of all objects that are stored in any of the caches. This will help us to detect quickly if an object is not cached anywhere.

With this approach, to add the information that an object is cached at a certain location, the object key is inserted into two bloom filters. The first one is the global bloom and the second is the bloom filter of the cache the object is added to. To look up an object location, the global bloom filter is queried first. If it returns a positive result, the c bloom filters are sequentially queried. As soon as a bloom filter returns a positive result, the cache to which this bloom filter refers to is accessed to fetch the object. If the global bloom returns a negative result, we know that the object is not cached anywhere, and we do not need to perform any queries.

The global bloom filter can reduce the number of lookups that we have to perform, in particular when the cache hit ratio is low, that is, there are many requests for elements

that are not in any cache.

Compared to Hashmap, our directory structure based on counting bloom filters requires less space as it does not store the actual object keys and their locations. However, it requires more time to determine object locations as it needs to perform several hash function calculations on both the global bloom filter as well as the local bloom filters. Please note that for convenience, we often call it only "bloom" instead of "bloom filter".

Optimized Bloom Filter If the global bloom returns a negative result, we know that the object is not cached anywhere. In the worst case scenario, the global bloom may return a positive result but the key is actually stored at the last queried bloom or in none of the locations because of the possibility of false positives in both the global and local blooms. In such a case, all of the c local blooms have to be queried. However, because each of these query operations requires hashing the key multiple times, equal to the number of the used hash functions, the total time required to check all blooms will be relatively high which adds up to the total request time.

Assuming that the caches all hold a similar number of objects, it makes sense that all the local bloom filters have the same size m of counters, and number k of hash-functions. Therefore, if we enforce that they all use the same k hash-functions, then, upon a query for a given key, we only need to calculate the values for each of the hash-functions once, and consecutively check whether the values are set for each of the local bloom filters.

Note that it would be possible to enforce that also the global bloom filter has the same size m as the local bloom filters by choosing an appropriate false positive rate for the global filter. However, this does not mean that it would have the same number k of hash functions, as the optimal number of hash functions does not only depend on m but also on n and the global bloom filter keeps track of many more objects than any local bloom filter. Thus, in our implementation, the global bloom filter works with a different set of hash-functions, and thus, will require extra calculations.

2.3.2 Choosing false positive rates for global and local bloom filters

For a single bloom filter, a false positive rate of p means that if we check for a key that is not in the set, the bloom filter will have all the counters for that key set to non-zero and return true with a probability of p.

The question now arises what is the false positive rate for our 2-level approach. Assume that the global bloom filter has a false positive rate of p_g and each of the c local filters has the same false positive rate p_l. Assume further that the global bloom filter has a different size than the local filters which, as we mentioned before, is a realistic assumption. As a result, its hash-functions have a different result domain and are independent of the hash-functions of the local filters. This means that the probability that a given key triggers a false positive on the global filter is independent of the probability that this key triggers a false positive on any of the local filters. Furthermore, as each of the local filters stores a different set of objects and the hash-functions generate uniform random distributions, the probability that a given key triggers a false positive on one local cache is independent of the probability that it triggers a false positive on one of the other local filters. Thus, for a key of a cached object to result in a false positive, it must first be a false positive for the global cache, which has probability p_g. Once that false positive occurs, the probability to trigger a false positive for any of the local filters is p_l. As there are c local filters, that second phase has a probability of $c * p_l$. Therefore, the overall probability of a false positive is

$$p_{all} = p_g * (c * p_l) \tag{2.3}$$

As such, given a desired maximum false positive rate p_{all}, we have to choose p_g and p_l so that the above equation is fulfilled.

One interesting observation is that the overall space requirement for all bloom filters is independent of the values we choose for p_g and p_l but only depends on p_{all}. This can be derived from equations 2.1 and 2.2. Assuming a total number of objects n and c local caches, then the global bloom filters holds n keys, and each local filter holds n/c keys.

Then the total number of counters, that is m counters for the one global filter and m' counters for each of c local filters can be calculated as

$m + c \cdot m' = \frac{n * ln(p_g)}{ln(1/2^{ln(2)})} + c \cdot \frac{n/c \cdot ln(p_l)}{ln(1/2^{ln(2)})} = \frac{n}{ln(1/2^{ln(2)})} \cdot (ln(p_g) + c \cdot ln(p_l)) = \frac{n}{ln(1/2^{ln(2)})} \cdot ln(p_g \cdot c \cdot p_l)$

Thus, as long as the product $p_g \cdot c \cdot p_l$ remains the same, the exact values of p_g and p_l do not matter, the space requirements remain the same.

2.3.3 Comparison using a global Bloom filter vs. not using a global Bloom Filter

In this section, we want to illustrate the advantage of having a global bloom filter compared to having only local bloom filters.

We can actually approximate a system that does not use a global bloom filter by having a global filter with a false positive of rate of $p_g = 1$ and a space requirement of $m = 0$, and thus, zero hash functions.

As the overall space requirements are the same as shown before, the local caches will be larger than if there is a global bloom filter, and the false positive rate of the local caches must be $p_l = p_{all}/c$, and thus, smaller than if there is a global bloom filter. With this, the number of hash functions without using a global filter is actually larger than when we use one.

Theorem 1. *Let k_{with} be the total number of hash functions when using the global bloom filter (i.e. p_g < 1), $k_{without}$ be the total number of hash functions when not using a global bloom filter and $c \geqslant 1$ be the total number of local bloom filters then $k_{without} \geqslant k_{with}$.*

Proof. From Equation 2.2 we can see that in case of a global bloom filter, the global bloom filter has $k_g = -log_2(\frac{p_{all}}{cp_l}) = -log_2(p_{all}) + log_2(c) + log_2(p_l)$ hash functions, and each local filter has $k_l = -log_2(p_l)$. In the case of no global filter, each local filter has

$k'_l = -log_2(\frac{p_{all}}{c}) = -log_2(p_{all}) + log_2(c)$ hash functions. With this, we have:

$$k'_l = k_g + k_l \tag{2.4}$$

Thus, $k_{without} = c \cdot k'_l = c \cdot k_g + c \cdot k_l$ whereas $k_{with} = k_g + c \cdot k_l$. When $c = 1$, $k_{with} = k_{without} = k_g + k_l$. Since k_g is always a positive number for any $0 < p_g < 1$, then for $c > 1$, $k_{without} > k_{with}$.

□

For example, assuming a system with a global bloom filter is configured to have 5 hash functions for the global bloom filter and 7 hash functions for each local filter, then a system with no global bloom filter that achieves the same false positive rate will have 12 hash functions for each local filter. Assuming 4 caches and thus 4 local filters, then $k_{without} = 48$ while $k_{with} = 33$.

Based on this, let's now look at the computational costs for checking whether a data item is in one of the cashes. If the data item is not cashed, by using a global bloom filter, we will only check this filter and then stop (unless there is a false positive, which is very unlikely). This means performing k_g hash calculations and then checking k_g bits in the global bloom. If there is no global bloom, we have to check each local bloom, each providing a negative results (unless there is a false positive, again very unlikely). If all local blooms use the same hash functions (optimized), then we have to perform $k'_l > k_g$ hash calculations. On top, we have to perform $c * k'_l$ bit checks. With a bit more calculations, the following is easy to derive.

Lemma 1. *Assuming a global bloom filter setup with k_g hash functions for the global filter and k_l functions each for the local filter, having an equivalent system without global filter will require k_l extra hash calculations and $c \cdot k_l + (c-1)k_g$ checking of bits in the filters if the item is not cached.*

For the previous example with $k_g = 5$, $k_l = 7$, and $c = 4$, checking for an input that is not in the cache will cause an extra 7 hash calculations and an extra 28+15=43 bit

checks if no global filter is used.

If the data item is cashed, when using a global bloom filter, we will first check that filter and then, on average, half of the local filters until we find the one with the match. This means performing k_g hash calculations and then checking k_g bits in the global bloom, then k_l hash-calculations (assuming all local blooms use the same hash function), and then $c/2 * k_l$ bit checks. This means in total $k_g + k_l = k$ hash-calculations and $k_g + c/2 * k_l$ bit checks. If there is no global bloom, we have to perform k_l' hash calculations, and $c/2 * k_l'$ bit checks. This means, the same number of hash calculations but still more checks. With a bit more calculations we can derive the following.

Lemma 2. *Assuming a global bloom filter setup with k_g hash functions for the global bloom filter and c local bloom filters, having an equivalent system without global filter will require on average $(\frac{c}{2} - 1) * k_g$ extra bit checks.*

For the previous example with $k_g = 5$, $k_l = 7$, and $c = 4$, looking up a cached item when not using a global bloom will cause an extra 5 bit checks.

Thus, deploying a global bloom filter is always beneficial but the effect becomes more pronounced if there is not a large cache hit ratio or if we cannot perform our optimized approach where we do hash-calculations only once and not for each of the local bloom filters. Because, then we have to do hash calculations for each local bloom until we have a match, and the local bloom filters have many more hash functions if there is no global bloom filter.

2.3.4 Experimental results

For our experiments, we use four caches which implies that the cache directory comprises of four local blooms in addition to the global bloom, and set the overall false positive rate to 10^{-3}. Using formulas 2.3, we set the global and local false positive rates to 0.0316 and 0.0079 respectively. We vary the total number of inserted objects across caches (local blooms) from 500k to 3 Mio. and evenly distribute the total number of

objects across caches. For instance, in case of 500k object, the global bloom holds information about the entire cache space of 500k objects while each of the four local bloom filters holds information about fourth of this number or 125k objects.

Table 2.1 shows all configuration parameters for both global and local blooms. Note that these configurations represent a single local bloom as all local bloom filters have a homogeneous configuration and we set the bucket size to 4 bits. The table shows that for each number of cached objects n, the respective number of buckets, m, in the global bloom is bigger than the local bloom due to the larger n. For the number of hash functions k, the global bloom uses less hash functions than a local bloom as it relatively has a bigger false positive rate. In addition, the table shows that as the number of inserted elements increases, the number of buckets increases and that is for both global and local blooms. Doubling the number of inserted elements, for example, doubles the number of buckets. This can be also explained by looking at formula 2.1, which shows that the number of inserted elements linearly affects the number of buckets. However, the number of hash functions, k, stays constant when increasing the number of inserted objects. This is because the number of hash functions, as formula 2.2 shows, is controlled by the ratio of m/n which stays constant as the number of buckets is linearly increased with the number of elements.

Space analysis: Figure 2.3a shows the size of the respective data structure with different number of cached objects that is only measured on one cache. Since both Bloom and Optimized Bloom use the same internal data structure and thus they both have the same storage size, we only show the memory size for the Optimized Bloom. The figure shows that for both approaches, the size of the directory increases as the number of the stored objects increases. However, in case of Optimized Bloom, such increase is much less. Adding half a million objects increases the bloom size only 4 MB while it increases the HashMap size with 20MB. In all cases, the figure shows that the HashMap requires almost 4 times memory space compared to bloom filter. This is due to the compact data array used for the former compared to the space costly HashMap data structure.

However, the respective memory saving depends greatly on memory size as well as average object size. To clarify it more, assume a cache with three GB of space, and

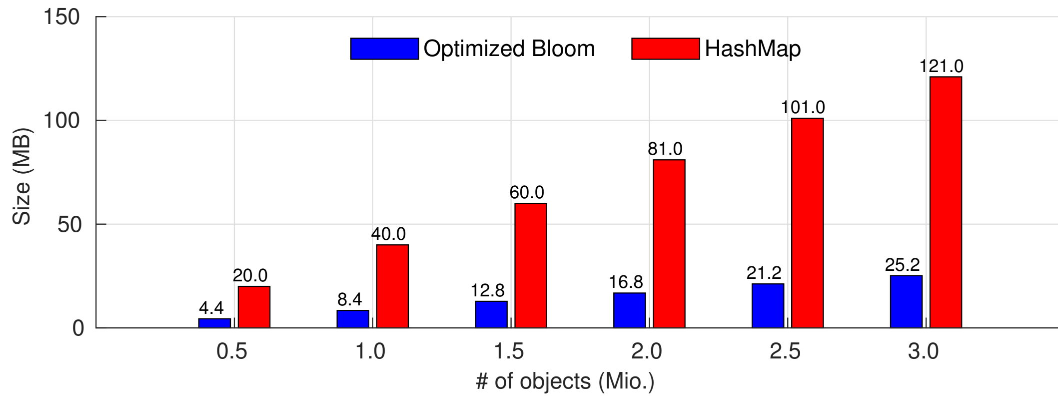
Table 2.1: Global and local bloom filter configurations for $p_{all} = 10^{-3}$, $N = 4$ and various n

number of cached objects (n)		**false positive rate (p)**		**number of buckets (m)**		**number of hash functions (k)**	
global	**local**	**global**	**local**	**global**	**local**	**global**	**local**
21500k	125k	0.0316	0.0079	3,595,147	1,259,461	5	7
1 Mio.	250k	0.0316	0.0079	7.190.294	2.518.921	5	7
1.5 Mio	375k	0.0316	0.0079	10,785,441	3,778,382	5	7
2 Mio.	500k	0.0316	0.0079	14,380,587	5,037,842	5	7
2.5 Mio.	625k	0.0316	0.0079	17,975,734	6,297,303	5	7
3 Mio.	750k	0.0316	0.0079	21,570,881	7,556,763	5	7

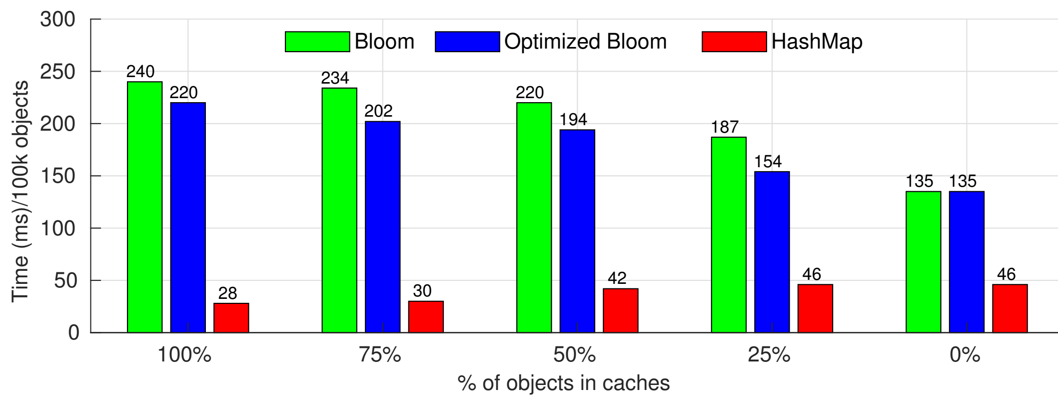
the average object size is 0.5 KB. Without cache directory, this cache can hold up to six Mio objects. While deploying Hashmap as a cache directory consumes around 121 MB of cache space, reducing cache capacity by nearly 250,000 objects, the bloom filter consumes only 25.2 MB, reducing capacity by only 50,000 objects.

Another appealing feature for utilizing bloom filter is that, unlike Hashmap, the object key size does not affect the bloom space size. In our experiments, we used a standard object key string that is generated by Hibernate and takes around 110 Byte. While having a longer object id, for example, will increase the Hashmap size, it does not increase the bloom directory size. This is because Hashmap stores the actual item key and its value (the cache server name) whereas bloom filter does not store the actual item but its hashed values, which stays constant regardless of the item's key size. The same observation can be also hold for the item's value.

Time analysis: Furthermore, we measure the processing time for querying objects on the three data structures of Bloom, Optimized Bloom and HashMap. Figure 2.3b shows the required time in milliseconds for querying 100k object keys for various cache hit ratios. The cache hit ratio indicates the percentage of the checked objects that are



(a) Size of the cache directory for various number of objects



(b) Query time for 100k objects for various percentage of cache hit

Figure 2.3: Optimized Bloom Filter Directory Performance

already stored in the cache compared to the overall objects checked. So a 100% cache hit indicates that all checked objects are stored in the cache while 0% cache hit indicates that none of them are stored. For all three approaches, the processing time for both Bloom and Optimized Bloom is much higher than HashMap, which is due to the fact that the item key has to be hashed several times for Bloom cases instead of hashing only once in case of HashMap. However, the optimized Bloom achieves faster processing time compared to the regular Bloom due to the optimization of reusing local hash function results in the former. The only exception where both bloom and optimized bloom have similar processing time is when there is zero cache hit which means that all items are eliminated by the first layer of global bloom and thus, the optimization technique is not

used anymore.

Another behaviour for both blooms is that the processing time decreases with a decreasing cache hit ratio which is caused by the presence of the global bloom. The more the number of non cached items, the more items the global bloom will eliminate and, thus, the less items the local blooms have to process. As a result, the processing time decreases. The last interesting behaviour is related to the processing time of the HashMap which, as opposed to its theoretical steady behaviour, is slightly increasing as the cache hit decreases. This can be explained by the working mechanism of the HashMap get operation that we briefly explained earlier. After hashing a key, the HashMap checks the respective bucket and if it contains multiple items, their keys will be sequentially scanned before returning a result. Thus, the lower the cache hit, the more buckets are scanned.

However, regardless if we use Hashmap or bloom filter directory, the time required for checking the item's location is negligible compared to the time required for accessing the actual item, either locally (0.1 ms) or remotely (1 ms). Therefore, the gain in using bloom filter is actually the space saving that is more important than the extra processing.

False positive analysis: Lastly, we validate the overall actual false positive rate for the above experiment, which is indeed, bounded by the configured value of 10^{-3}. This ratio is based on the final number of false positives generated by any of the local blooms that have also been falsely identified by the global bloom.

In summary, these experiments show that using Bloom Filter as a cache directory induces a negligible extra processing time, but has a considerable advantage of memory saving.

2.4 AdaptCache: Adaptive Cooperative Cache Framework

The Adaptive Cooperative Cache Framework or AdaptCache, extends the CC by adding an external component, called the Analyser [74], that is able to receive messages from

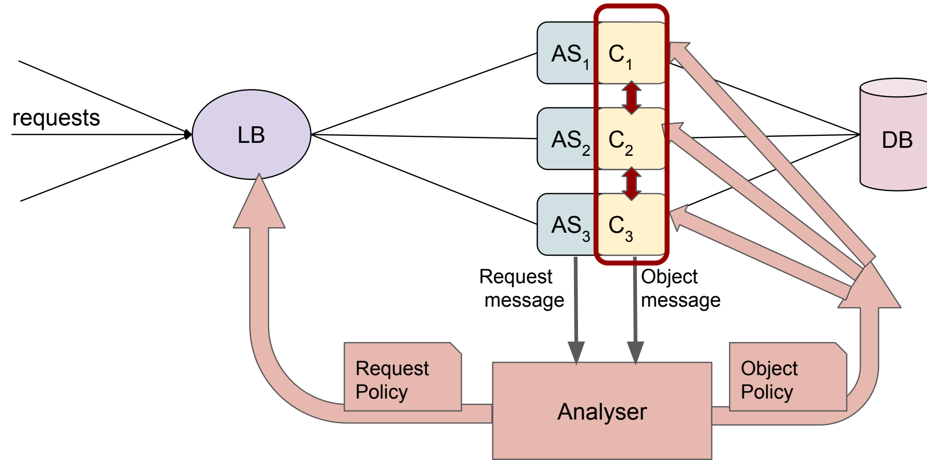


Figure 2.4: Adaptive Cooperative Cache Framework

ASs regarding their requests as well as their respective objects and, based on these messages, distributes load across servers and objects across caches. The original implementation of the extended CC was static whereby the Analyser received a workload trace and based on these traces decided on a system configuration. For this PhD thesis we have extended this framework to work in a dynamic environment. In this section we present AdaptCache in its extended format.

2.4.1 Request- and Object Policies

The architecture of AdaptCache is shown in Figure 2.4. In order to distribute load (requests) and objects, the *Analyser*, which is the core component of the adaptive cache framework, generates two types of policies, the *RequestPolicy* and the *ObjectPolicy*. The RequestPolicy contains rules that tell the load-balancer how to distribute requests among the AS instances while the ObjectPolicy tells the individual caches which objects they should cache locally. Both policies are generated dynamically by observing the most recent requests and the objects they access. In general, the goal of the generated policies is to assign a request to the server that has most of the objects accessed by the request in its local cache to avoid calls to remote caches. The overall goal is to reduce request execution time, application server load, and the amount of messages exchanged over the

network. While this thesis reuses the Analyser component developed in [74], we extended it to work in a dynamic environment and developed completely new algorithms to generate the policies.

2.4.2 Logging

In order to generate policies, AdaptCache needs to observe workload behavior. At each AS, AdaptCache generates for each request received a request log containing information such as the URL and the server identifier. Additionally, it generates object logs whenever a request accesses an object, keeping track of the object identifier and the request type that accessed it. These logs represent the workload meta-data and are sent to the Analyzer for further processing.

2.4.3 Policy Deployment

The Analyzer generates new Request- and ObjectPolicy based on the most recent meta-data information whenever the current configuration does not serve anymore well the current workload. In the next chapters, we discuss in detail a set of algorithms that generate these policies. The new policies are sent to the load balancer and the cooperative cache instances where they replace the current instances of the corresponding policy files. Note that the policy deployment was introduced in [74] and for the purpose of this thesis, we have refined it to work in a dynamic environment.

The load-balancer uses the RequestPolicy to distribute the requests. The request policy contains entries for request types (i.e. URLs) and the servers to where to send a request instance of that type. Upon receiving a request of a certain type, it checks in its current policy file whether an entry exists for that particular request type. If yes, the request is sent to the AS indicated in the entry. If not, the load-balancer falls back to a default algorithm, which in our case is round-robin. In our implementation, we extended the load-balancer provided by Apache[3] 2.0 to be able to receive the RequestPolicy files during runtime and distribute requests accordingly. The ObjectPolicy files are sent to the

[3]http://httpd.apache.org/

local caches to tell them which objects to store locally.

2.4.4 Lazy Data Migration

Object migration according to the ObjectPolicy is done in a lazy fashion. The ObjectPolicy identifies the objects that the local cache should maintain. With this, there are two types of caching: (i) an object that does not appear in any ObjectPolicy file is loaded into the local cache of the first AS to request it; (ii) objects that appear in the ObjectPolicy file of one cache will be maintained by this cache on a long-term basis. When a local cache receives a request for an object from its AS, it checks its ObjectPolicy. If the object identifier is contained in the policy and it is already in the local cache, it is simply returned to the AS. If it is not in the cache, it is requested for migration from a remote cache (checking the cache directory) or it is retrieved from the database; it is then cached locally, and returned to the AS. When a remote cache receives a request to transfer an object, it does so, discarding the object locally. That is, objects are not immediately migrated from cache to cache when a new set of ObjectPolicies is created but are moved lazily whenever they are requested. Thus, the timing of the migration is driven by application access patterns, and objects that are no longer accessed will not be migrated at all.

However, even if done lazily, migrating objects across caches while the system is up and running puts extra load on both the source and destination servers, which as a result impacts the overall performance [34, 43]. Particularly, one can expect that the performance drops at the beginning of a reconfiguration as requests are redirected to different servers and data migration starts, resulting in a lower local hit rate. But then, hopefully, performance quickly improves once data is newly distributed. In the next chapter, we will show how we mitigate such migration impact by assigning partitions to caches that already have most of their objects, thus decreasing the number of objects to be migrated.

Table 2.2: Request and object logged information

Request Information	**Object Information**
Request URL	Object ID
Server ID	Server ID
UUID	Operation Type (get, put)
	Access Type (local, remote, db)
	UUID

2.4.5 Workload Meta-data

In this section, we have a closer look at the actual data collected. In order to keep the logging overhead small, AdaptCache uses two different tracking and analysis phases. In the coarse-grained tracking phase, AdaptCache only extracts information from the running system that is sufficient to detect workload changes that require to repartition or readjust replication. Once the Analyzer determines that a reconfiguration is needed, fine-grained logging is activated and much more detailed information is logged that will help to find appropriate new Request- and ObjectPolicies as well as the right replication level for each object. Originally [74], AdaptCache performed only fine-grained tracking. We added the coarse-grained logging phase in order to reduce overhead.

Fine-grained Phase In the fine-grained phase, AdaptCache generates detailed information about the currently executed requests and their respective objects, similar to [74]. This logging information is generated at the AS and the cache and then sent to the Analyzer. In particular, AdaptCache generates a request log for each request instance the AS receives with the information depicted in Table 2.2. As mentioned in Section 2.1.1, the request URL identifies the request type. In order to be data-aware, a request type identifier is the entire URL, including the specific method that is called and the list of the parameters that are the input for the method. The UUID offers a unique identifier for each individual request instance. There can be many request instances for a given request

type/URL.

Additionally, AdaptCache generates an object log whenever a request accesses an object, keeping track of the object identifier, the server to which the request was submitted, the operation type (put/get) and the access type (object found in the local cache, in a remote cache, or had to be retrieved from the database), and the UUID of the calling request instance.

AdaptCache uses an interceptor approach to generate these logs that does not require changes to or knowledge of the application nor the application server code. These logs represent the workload meta-data during the fine-grained logging phase and are sent continuously to the Analyzer. The fine-grained phase lasts for a configurable time, called the log window size. The Analyzer processes the logs sent during the log window size and reformats them into summary information such as the number of accesses per request type, number of accesses per object, etc. The details will be discussed in the following chapters. The Analyser then generates the new policies and possibly new replication configuration. These new policies are then sent to the load-balancer and the cooperative cache instances where they replace their current policy files, and then the system switches back to the coarse-grained logging phase.

Coarse-grained Phase In the coarse-grained phase, much less detailed information is needed. In fact, as we will discuss in detail in the next chapters, the local hit ratio, which indicates the ratio of objects accesses where the object is found in the local cache over all object accesses, and the ratio of write vs. read operations are the two metrices that are important for determining whether reconfiguration is necessary.

Both information types can be easily retrieved from the object logs. Thus, during the coarse-grained phase, AdaptCache only generates simplified object logs that only indicate the operation type and the access type and sends those to the Analyzer. No object identifier or UUIDs need to be determined, reducing the logging overhead at the AS and the cache considerably. Furthermore, the Analyzer has also considerably less overhead during the coarse-grained phase to process these logs to determine the hit ratio and the read/write ratio.

3

Adaptive Object Partitioning and Migration

In this chapter, we show the various request and object partitioning solutions that we have developed to distribute requests and objects across server caches with the aim to minimize remote cache access and balance the load. We first start by presenting our workload analysis for typical e-commerce queries that inspired our dynamic partitioning solutions and also helped us in extending our evaluation benchmarks. From there, we present a suite of request and object partitioning algorithms and show how we tailor them to work in a dynamic workload environment. Finally, we show experimental results using various workloads.

3.1 Workload Analysis

A holistic caching solution has to be able to handle complex, real-life workloads. We have had a closer look at eBay[1], a well known e-commerce application. eBay allows users to sell and buy items online: sellers can post items, and clients can search the posted items using various filtering conditions and sort the results based on different sorting criteria. The goal of analyzing this application is to help us understand the main characteristics of requests and the objects they access as typically found in online applications.

3.1.1 Request Characteristics

In eBay, a buyer typically first chooses a coarse category (fashion, electronics), and then a subcategory. Furthermore, more advanced searches can be executed that restrict the price, location and other attributes. A general key word search is also possible. The first result page contains, by default, 50 items and the user has to explicitly request each of the next result pages one by one.

Request Overlap

As we mentioned in Section 2.1.1, a request type in AdaptCache is a URL including method parameters. Thus, it is obvious that request types overlap in the objects they access. In eBay, the first result page of a request searching for HP laptops might overlap with a request that searches for only new laptops. The fact that different request types do not access disjoint data sets makes data partitioning a challenging task and has been considered by previous work [39, 26, 87].

Request Popularity

The most common search on eBay is to simply look within a subcategory, possibly restricting some additional attributes. Thus, one can expect these request types to be quite

[1] https://www.ebay.com/

frequent. Additionally, even a general keyword search appears to be skewed because after typing the first keyword, the system suggests extensions that represent searches in the recent past. By suggesting them to the next user, it is quite likely that their popularity will remain high. Furthermore, a per-user search history is maintained. All these features likely lead to a highly skewed request popularity and favor caching solutions.

Request Drift

When data changes frequently, e.g., due to inserts, search requests will change their results over time, which we refer to as *request drift*. For example, assume that a user wants to search for an item within a certain category but prefers the returned result to be sorted based on listing time (i.e., newly listed items appear first). In this case, the set of items returned when the same request type is submitted twice, thus given two request instances of the same request type, will not be identical if items of the searched category are inserted in between the two calls.

To understand how prevalent request drift is on eBay, we conducted the following experiment. We asked for the first result page of a search including a specific subcategory. For this simple search, there are 12 request types, one for each sorting order that eBay offers. For a period of 60 minutes in total, we submitted a request instance of each of these request types every 10 seconds. Thus, at the end of the experiments, we have 360 result sets of 50 items each for each request type.

To track how these result sets change over time, we look at the accumulative request drift over the observed time period. Let DS_i be the accumulative data set containing all data items that were returned since the start of the experiment up to the i'th call. As every individual call or request instance returns only 50 items, the size of DS_i is a measure of the accumulative drift.

Overall, we could categorize the 12 sorting orders offered by eBay into three groups according to the drift rates: large, medium, and small. For illustration in this thesis, we pick the sorting orders *EndingTimeSoonest, StartTimeNewest* and *BestMatch* to represent each group, respectively.

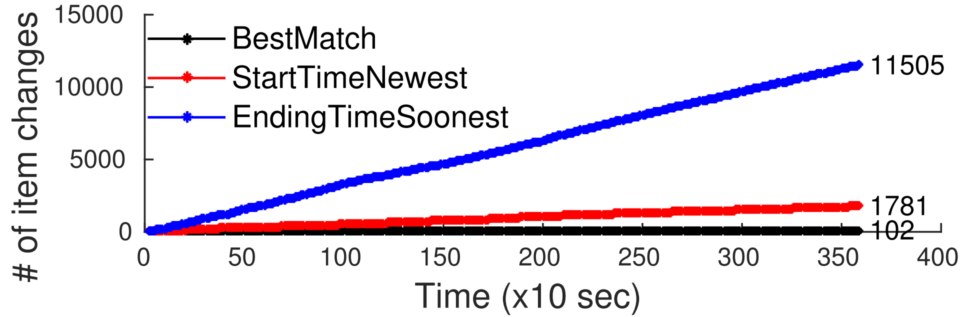


Figure 3.1: Accumulative request drift over time

Figure 3.1 depicts the accumulative request drift over time. For all three request types, the result set drifts. This drift is very small for BestMatch (only 102 different items in one hour), higher for StartTimeNewest (few thousand) and very high for EndingTimeSoonest (more than 10,000).

In BestMatch, the items are sorted based on criteria such as the seller's rating and the item's popularity. These factors do not change quickly over time. Thus, a popular item will likely stay popular for some time. It can disappear from a result set because it is sold or because an item that scores higher in the BestMatch scoring function is inserted.

The other two search orders sort items based solely on time attributes (start/expiry). This causes the results to change at a higher rate because of the high insertion and extremely high bid expiration rates. eBay allows users to sell their items in one of two styles: "auction" and "buy-it now". Both tag items with an expiration date. If an auction item expires, then it won't appear again in future search results. We can expect those items to expire at the same rate as new items are created. For buy-it now items, in contrast, eBay allows users to keep their items active until they are sold. eBay then automatically renews the expiration time every 30 days until it is sold. Thus, a single inserted item can expire many times, leading to a very large accumulative drift when items are sorted by their expiration time.

In summary, request drift can have a significant impact on the dynamicity of the workload, and requires a continuous adjustment of data partitioning strategies, even if

the request types themselves do not change. Additionally, if a request type has quickly changing result sets, the potential for caching becomes restricted.

3.1.2 Object Characteristics

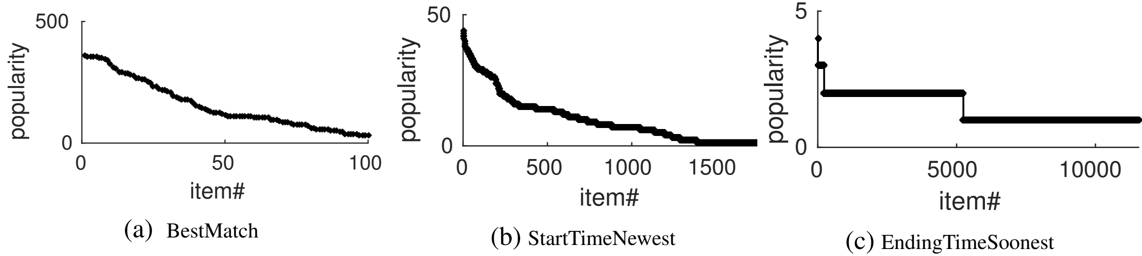
Let's now focus on how individual objects are accessed. Just as with requests, we can envision not only differences in popularity of objects but also that the popularity changes over time, thus showing temporal locality.

Object Popularity

Figures 3.2(a-c) show for each of the three sorting orders and for each of the items they access, how often the item is accessed, sorted from most popular to least popular item. Please note that the y-values are quite different for the different sorting orders. In BestMatch, the most popular item is returned in all 360 requests that were executed, and from there popularity decreases very slowly. Most objects are fairly popular with half of the objects accessed more than 100 times. This behavior can be expected when only a small number of objects are accessed overall. In StartTimeNewest, the most popular item is accessed fairly frequently (close to 50 times) but popularity then decreases more quickly following a more skewed distribution. With EndingTimeSoonest all items show nearly the same very low access frequency. The reason is that at any given time, there are many items that are close to expiring. With making calls every 10 seconds, many of the items that were returned in a first call have already expired at the time of the second call, and are no longer in the result set.

Object Temporal Locality

Another important aspect is an item's temporal locality. An important assumption for caching to work well is that items that are currently accessed will be accessed again in the near future. At some time point, however, they might become unpopular, and once they haven't been accessed for a while it becomes unlikely that they will be accessed in the near future. Thus, the caching solution has to get rid of them.



(a) BestMatch (b) StartTimeNewest (c) EndingTimeSoonest

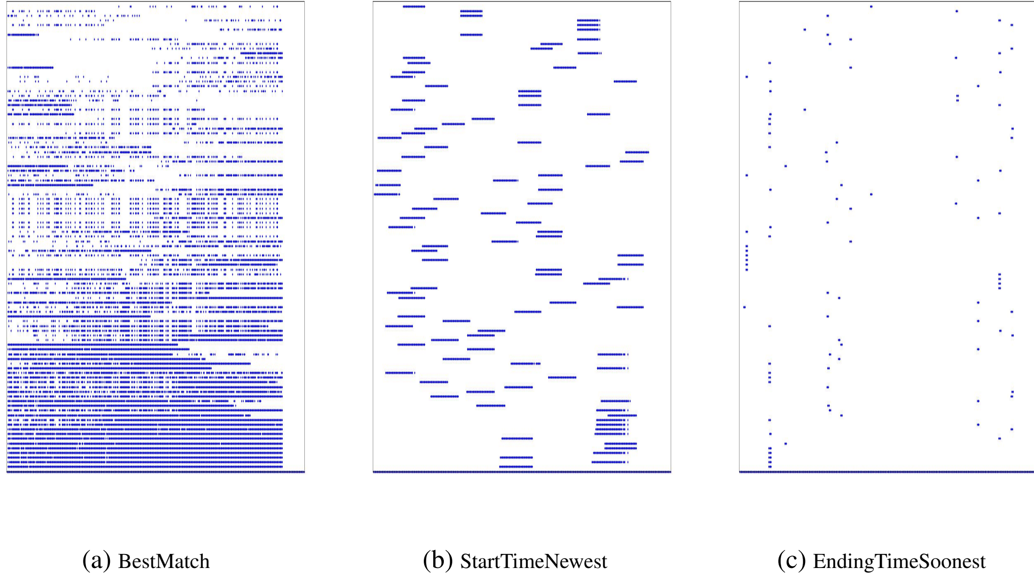
Figure 3.2: Item popularity

Figures 3.3 (a-c) show that this assumption holds for all sorting orders. Each figure depicts the 100 most popular items on the y-axis with decreasing popularity. The x-axis depicts time. There is a dot for an item x at time i, if x was contained in the result set of the ith call of that request type. As expected, for BestMatch we can see that the most popular items were returned in every call. Others appear in the first calls but not in later ones, likely because they were removed. Some only appear in later calls, likely because they were inserted sometime during the experiment. For StartTimeNewest, an item becomes popular at the moment it is inserted, then is in the result set for several calls before it disappears from the result set because new items were created and replace the older ones in the result set. The behavior for EndingTimeSoonest is similar, but each item is only in the result set for very few times before it is removed because its ending time has expired (or was reset).

In summary, we can confirm that temporal locality is generally given, which is an attractive property for caching if handled properly. However, for some requests and objects, the time window is very short; the benefits of caching for them is less clear.

3.1.3 Further Analysis

In addition to what we described above, we conducted experiments with different categories, and additional filtering conditions such as price. We also ran the experiments at different times of the day. In all cases, we found similar patterns. Only when we narrowed down the search by, for example, searching for rare items, we did not notice significant difference between the drift rates of different searching orders because all of



(a) BestMatch (b) StartTimeNewest (c) EndingTimeSoonest

Figure 3.3: Temporal Locality

them returned very small data sets. Thus, we believe that a large set of requests posted to eBay follow the patterns we have discussed in this section, and our caching solution is designed with these patterns in mind.

3.2 Policy Generation

In this section we discuss in detail the algorithms we have explored to distribute requests and objects across application servers. The main goal of these partitioning algorithms is to distribute end-user requests as well as their respective data among different cache nodes so that the overall number of distributed calls is low and the load is equally distributed among different cache nodes.

Our approach was inspired by previous approaches, explicitly SCHISM [26] and Sword [91], developed in the area of distributed database systems. They aim in distributing database objects across storage nodes such that most transactions only need to access one data store, and thus, only few distributed transactions are required. This avoids the

costly 2-phase commit protocol. In order to distribute objects, such that objects that are accessed together by a transaction end up at the same partition, they have proposed building an object-based workload graph. They map each data item in the workload to a vertex in the graph. In some cases, they give each object a weight equal to its access frequency in order to consider the load created by each object. Alternatively, they give all items the same weight in order to keep the amount of data stored at each data store equal. The approaches differ in how edges are constructed: SCHISM builds a simple graph while Sword builds a hyper-graph. Once the workload graph/hyper-graph has been constructed, a graph partitioning tool that builds partitions is used so that the number of edges between objects that are in different partitions is minimized. This helps in putting objects that are accessed together (have many edges) into the same partition. This is similar to what we want to achieve. For our application, that means if two objects are accessed by the same request (and thus have a connectivity edge) the partitioning tool tries to assign them to the same partition. The partitions are then distributed across nodes.

Our first solution space, which is object distribution first, is based on this approach. It generates the object partitions which basically represents the object policy as described in Section 2.4. However, we need much more because we look at very different challenges. Firstly, we do not only need to decide which objects to put into which cache, but we also need a request policy that decides which application server executes which requests in order to achieve optimal access locality. Furthermore, as we have shown in the previous section, the workload is highly dynamic and even requests that remain popular for a long time tend to change their accessed data set over time. Additionally, new requests that specifically query the newly inserted data appear while old requests that query the excluded data disappear. Thus, our approach has to be highly dynamic, not only continuously adjusting how we partition the data but also how we distribute requests across the application servers.

Therefore, we explore several ways of building a graph as well as performing partitioning to better understand the trade-offs. More precisely, we do not only look at an object graph but also at a request graph, where requests are nodes that are connected when they access common objects. With this, we are likely to assign requests that access

common objects together into the same partition. Furthermore, we have a close look at how to assign the partitioned data to servers. Finally, in each iteration, we determine requests and objects that can be pruned, that is, ignored for distribution. This may be because they have fading popularity or because the requests have too high drift rate to benefit from caching.

3.2.1 Parameter Collection

We have briefly illustrated in Section 2.4 that the Analyser does not trigger policy generation on a regular interval basis; rather it does so only when the currently deployed policies do not serve the current workload anymore. And in order to detect workload changes the Analyser uses the coarse-grained phase while it uses the fine-grained phase to collect detailed information about the current workload that causes such a change.

Coarse-grained Tracking

In the coarse-grained tracking phase, the Analyser monitors the workload to measure the performance in order to decide about triggering reconfigurations. The question is how to measure performance. We could use the throughput and evaluate throughput degradation. However, throughput might decrease simply because the overall load submitted to the system might decrease (e.g., night time). Thus, we choose to use the local hit rate assuming this to be a more stable performance metric.

In order to decide when to trigger policy generation, the Analyser periodically measures the current local cache hit ratio h_c. To do so, the Analyser maintains three counters for local, remote and db accesses and whenever an object log arrives, the respective counter is increased by one. The local cache hit ratio is then calculated by dividing the number of local accesses by the total number of all accesses. Recall from Section 2.4.5 that during the coarse-grained phase, simplified object logs are sent.

To determine whether reconfiguration is needed, the Analyzer also needs to keep track of the maximum local cache rate h_m since the last time new policies were installed. Once h_c has degraded considerably compared to h_m, or more precisely, once

Table 3.1: Collected parameters for request and object partitioning

n_p	number of partitions
$r.O$	set of objects that request type r accessed in the last window
$r.n$	number of accesses of request type r
$r.n_o$	number of accesses of request type r to object o
$o.n$	overall number of accesses to object o

$\frac{h_m-h_c}{h_m} > threshold$, the Analyser switches to the fine-grained phase where fine-grained information related to requests and objects will be gathered. That means, reconfiguration is not triggered on a fixed interval basis, but varies depending on how much workload changes influence the performance. However, such an approach does not trigger a reconfiguration when hit rates do not get worse. Nevertheless, it might be possible that with a better configuration, hit rates could be even better. In order to not miss these situations, we trigger a new reconfiguration after a certain, fairly long time, even if the degradation threshold has not been reached.

Fine-grained Tracking

As a first step of reconfiguration the system switches to fine-grained tracking. In the fine-grained tracking phase, the Analyser utilizes the detailed request and object log messages as described in Section 2.4.5 to generate statistics for each request and accessed object. These statistics are generated for a predefined observation interval and are used as input for the partitioning approach. Table 3.1 illustrates the required parameters collected for our request and object partitioning solutions. The first parameter is the number of partitions, n_p, which also represents the number of AS. The partitioning solutions use this parameter to partition the workload graph into a number of partitions equal to n_p. In addition, the Analyser generates statistics per request and object. Particularly, for each request type, represented by the request identifier r (URL), the Analyser keeps track of the set of objects the request instance of this request type accessed in the last window, or $r.O$. It also calculates the total number of instances of request types, i.e. how often

request r was requested by some clients, denoted as $r.n$. In addition, for each request type r, the Analyzer needs to know how often its instances accessed an object $o \in r.O$ denoted as $r.n_o$. Finally, for each object o, the overall number of accesses to that object, or $o.n$, is calculated.

In the following, and for simplicity of description, we use request r instead of request type r. For instance, when we say that the Analyzer assigns a request r to a server in the RequestPolicy, it means that all request instances of request type r will be executed at that server.

Note that all of these parameters only consider read requests as all of the partitioning solutions we propose only consider read requests. This is due to the fact that partitioning write requests across ASs will not have real performance benefit as all write requests have to access the back-end database anyway; which becomes the significant performance impact. Therefore, the location of executing write requests is not as critical. However, when considering replication in the next chapter, we will distinguish between reads and writes, and thus, nomenclature will be slightly different.

Another important parameter that we briefly discussed in Section 2.4.5 is the log window size. It represents how much workload meta-data from the fine-grained phase should be taken into account for the new policies. If too much is used we could consider objects and requests that are no longer relevant; if we take too little, we can miss important patterns. In the evaluation section we analyze the impact of the log window size in more detail.

3.2.2 Object Distribution First

In this first solution space, we first assign objects to servers, that is, we build the ObjectPolicy, and then build the RequestPolicy with the goal to distribute requests among the servers so as to achieve a high local hit rate.

Object-based Graph

Our object-graph is built as in SCHISM [26]. What we consider differently is that we build the graph using request and access frequencies that were collected within our log window as described in Section 2.4.5 while SCHISM considers the log of a past execution. In SCHISM, vertices are objects and two vertices are connected if there is a transaction that accesses both data items. The edge weights represent the number of transactions that access both objects. Vertices have a weight that either reflects the total number of accesses to that vertix or the size of the object. The graph is then split using a k-way min/cut algorithm that splits the graph into n_p object partitions op_i, $1 \leqslant i \leqslant n_p$, such that each partition contains almost equal vertix weight, at the same time, the total edge weight that is cut, i.e., where the edge vertices reside in different partitions, is as small as possible. Cuts through heavy-weight edges should be avoided as they lead to many distributed transactions that must access more than one partition. An example of such a k-way min/cut algorithm is the multi-level graph partitioning algorithm proposed in [55] and implemented in METIS[2], a well known graph partitioning library.

In our caching solution, we use the number of accesses to an object, $o.n$, as vertex weight to guarantee equal load, and we use a slightly different edge weight to keep track of request drift. Although a request type might access two objects, not every request instance of this request type might access these two objects. Therefore, we use as edge weight the sum of both objects' number of accesses caused by request types that access both objects. For instance, assume only request type r accessed both objects o_1 and o_2 and it accessed o_1 x times, and o_2 y times within the observation window, that is $r.n_{o_1} = x$ and $r.n_{o_2} = y$. Then, we set the edge weight to $x + y$. Thus, the more often the objects were accessed, the larger the edge weight, and the more likely both objects will be in the same partition op_i. The RequestPolicy will then have request types assigned to the node that holds partition op_i and all request instances of this request type, whether they only access one of the objects or both, will trigger only local access. We refer to this approach as *Object-based Graph Partitioning*, or **Obj-GP**.

[2] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

Object-based Hyper-Graph

Instead of building a graph, Sword [91] uses a hypergraph. Objects are vertices, and all objects that are accessed by the same transaction are connected through a hyper-edge whose weight is the number of times the transaction is executed. Our *Object-based Hyper-graph Partitioning* approach, referred to as **Obj-HgP**, follows the same approach connecting all objects accessed by a specific request type with a hyperedge with the edge weight being the number of times the request type executed during the observation period.

Both graphs can have a large number of nodes as there are many objects in the cache. Furthermore, the standard object graph can have many edges. The problem is aggravated due to the request drift which can cause an edge explosion. The hyper-graph has less edges but each edge will connect many objects if there is a large request drift. Furthermore, it cannot reflect how many times each of the objects is accessed by the request type as there is a single edge for a request type.

A further important restriction of this approach is that we have difficulties considering cache size. One cache might get very few but very frequently accessed objects, another might get many infrequently accessed objects. In order to have an upper cap on the amount of objects (or total object size) stored at a server, we could use object size instead of access frequency for each object vertex. However, then load-imbalances might occur. Alternatively, we would need further conditions on the graph (e.g., two-valued vertex weights).

Request Distribution

To route queries among servers after partitioning the data, both Sword [26] and SCHISM [91] leverage the query translation capability of a front-end database system that extracts the respective objects for each transaction. The transaction is then (partially) executed at the affected partitions. If there is more than one, the transaction is distributed.

In contrast, our load-balancer will send the request to a single AS where it is executed

in its entirety and that AS will then retrieve the objects from the local cache, remote caches and the database as needed. Thus, we create for each object partition op_i, $1 \leqslant i \leqslant n_p$ where n_p is the number of partitions, a request partition rp_i so that the RequestPolicy will assign all requests in rp_i to the AS whose cache maintains op_i. The idea is that if a request is assigned to request partition rp_i then it will find most of its objects in object partition op_i.

To this end, we develop a gain formula that determines the gain of assigning a request type to request partition rp_i by summing up all its object accesses into op_i during the last observation window. We then assign a request to the partition with the highest gain. Formally, let $r.O$ be the set of objects that request type r accessed during the last observation window. Let $r.n_o$ be the number of times r accessed an object o during that window. Then the gain to add r to rp_i is expressed as

$$gain(r, rp_i) = \sum_{o \in op_i} r.n_o \mid o \in r.O \tag{3.1}$$

3.2.3 Request Distribution First

The second solution space assigns first requests to servers, that is, builds the RequestPolicy, and then builds the ObjectPolicy.

For that, we build a request-based graph that maps each request type to a vertex in the workload graph. As one of our partitioning goals is to assign equal load to each partition, we assign to each vertex as a weight the number of times the request type was executed in the last observation window multiplied by the average number of objects the request type accessed every time it was executed. The average number of objects accessed by request r, or $AvgObjNo(r)$, can be calculated as

$$AvgObjNo(r) = \frac{\sum_{o \in r.O} r.n_o}{r.n} \tag{3.2}$$

Request-based Graph

To achieve the second partitioning goal of collocating requests that access common objects on the same machine, we add an edge between two vertices (requests) if they access at least one object in common. The edge weight reflects how often they access common objects, i.e., it is the sum of all accesses of these two request types to objects that both request types accessed in the last observation window:

$$weight(edge_{r_1,r_2}) = \sum_{o \in (r_1.O \cap r_2.O)} r_1.n_o + r_2.n_o \tag{3.3}$$

The more often they access common objects, the less likely the requests end up in different partitions. Objects that are accessed by only one request do not create any edges.

We use two mechanisms for partitioning this request-based graph. The first again uses a k-way min/cut partitioning algorithm, and we refer to this approach as *Request-based graph partitioning* or **Req-GP**.

The second attempts to simplify and speed up the partitioning approach. It first traverses the graph using Breadth First Search (BFS) starting with the most popular request and builds an ordered list of vertices. The heuristic behind using BFS is to encourage adjacent vertices to remain close to each other. After traversing the graph using BFS, we use the traversed list to assign requests (vertices) to different partitions using bin packing. However, we set a threshold for the capacity of each partition as being the sum of all vertex weights divided by the number of partitions plus a small variation. By doing so, we guarantee that the workload will be evenly distributed among different partitions. We refer to this simple heuristic approach as **Heur**. We decided to include this simple partitioning scheme into our evaluation to see whether sophisticated and costly graph partitioning algorithms really provide a significant advantage.

Request-based Hyper-Graph

A further alternative is to build a hyper-edge for each object that connects all requests that have accessed that object at least once. In this case, the weight of the hyper-edge will be the access frequency of that object. We use the standard hyper-graph partitioning algorithm and refer to the approach as **Req-HgP**.

Object Distribution

After having generated the request partitions, we now have to create an object partition op_i for each request partition rp_i. Similar to the gain function we created before, we use a gain function to determine the gain we achieve by adding an object o accessed in the last observation window to the object partition op_i pertaining to request partition rp_i. Our gain function expresses our expectation of how many local hits this assignment will produce. We then assign o to the partition that produces the maximum gain. More formally, we calculate the gain as

$$gain(o, op_i) = \sum_{r \in rp_i} r.n_o \mid o \in r.O \tag{3.4}$$

Additionally, we also want to make sure that we do not add more objects to a cache than the cache capacity allows. In our current implementation we assume caches have similar sizes and objects to have the same size. With this, we should assign roughly the same number of objects to each cache. We do so by assigning objects in descending order of their access rates. Thus, the more popular the object, the more likely it will be assigned to the partition with the highest gain. Once a partition reaches its object threshold, it can no longer host objects and thus, will not be considered for the assignment of subsequent objects. If cache sizes or object sizes differ, we can simply build thresholds and assignments determined by cache and object sizes.

3.2.4 Assigning Partitions to Servers

New policies are sent to the load-balancer and the object caches whenever needed. However, if we assign each pair rp_i/op_i of request and associated object partition to a random server, it might be quite likely that most requests (and the objects they access) are now assigned to a different server compared to the previous policies. As a result, when a request first executes at the new server, its objects, although according to the new ObjectPolicy should reside at the local cache, are still probably located at a different cache and have to be migrated to their new location. Despite the fact that this happens lazily the first time the object is accessed, it can still generate a considerable overhead.

Thus, the idea is to keep the number of objects to be migrated low by looking at each cache's local content and find the partition assignment that offers a maximum overlap of current cache content and new object locations.

In a first step, we determine for each object partition op_i and server S the $overlap(op_i, S)$ as the number of objects in op_i that already reside at S's cache. We then determine for each possible assignment of object partitions to servers the gain in number of objects that do not need to be migrated because the servers' caches already hold these objects. We then choose the assignment with the largest gain.

Figure 3.4 shows an example where three partitions need to be assigned to three servers each of them already having certain objects in their caches. The left table shows for each server and partition the number of objects that already reside on the server (overlap). For instance, three of the objects of object partition op_1 already reside in the cache of S_1. From there, the right table shows for each of the 6 possibilities to assign partitions to servers, the overall number of objects that do not need to be migrated (sum of the corresponding overlap values of the left table). In this example, the assignment $op_3 \rightarrow S_1$, $op_2 \rightarrow S_2$, $op_1 \rightarrow S_3$ gives us the biggest gain.

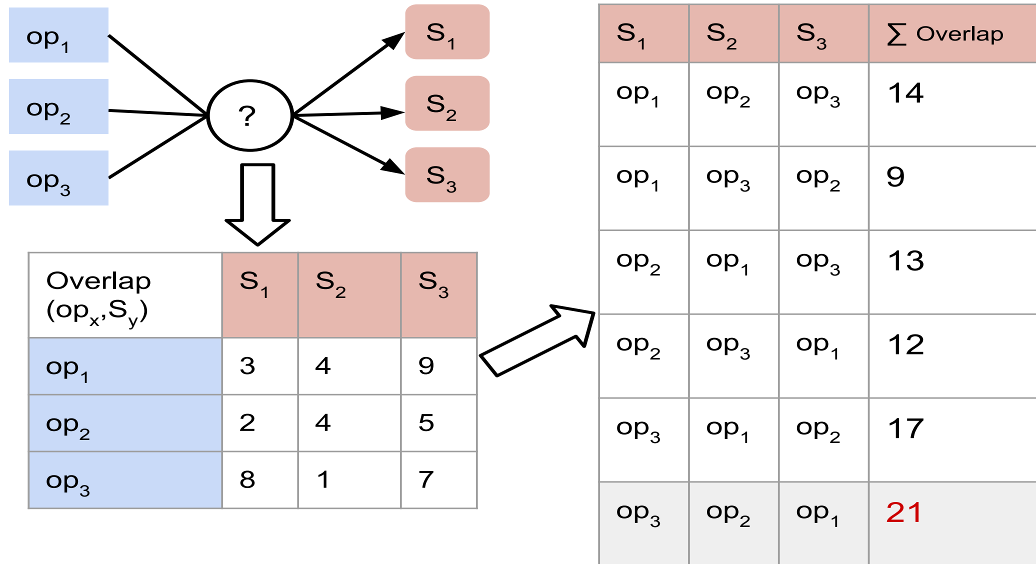


Figure 3.4: Assignining partitions to servers example

3.2.5 Reducing Workload Meta-data

It is important to prune the logging information to be used for generating the workload graphs for two reasons. First, the more information we use the longer it will take to generate and partition the graph. Second, as we discussed previously, not all request types are worth to be considered for partitioning as for some of them there might be little benefit for caching their objects. Even worse, if considered, they might trigger objects to be migrated during reconfiguration, even having a negative impact on performance.

There are two types of requests that we consider to ignore for our graph generation. The first are those that had low weight during the last observation window because they were not called frequently. The second type are requests with high drift as discussed in Section 3.1.1. For that, we tag each request with a *drift rate* value in the range [0,1] which represents the average rate at which a request changes the objects from one call to the next. For instance, if a request returns 50 objects on average every time it is called, and on average two objects are different from one call to the next, the drift rate is 2/50 or 4%.

After having decided on the request types to be eliminated, we also ignore all objects that are only accessed by the requests that we discarded.

3.3 Evaluation

In this section we show the evaluation results for our caching solution using the YCSB and RUBiS benchmarks. We conducted our experiments on a cluster of 9 nodes: one each for load-balancer, database server, analysis server and client emulator, and four for application server instances. Each of the machines has an Intel(R) 2.90GHz Dual-core, 8GB of RAM, and runs with Linux OS. In the four application server instances, we dedicate one GB of RAM to the cache while the rest is used by the business processes of the application server.

3.3.1 Benchmarks

YCSB: The Yahoo! Cloud Service Benchmark has been developed to evaluate key/value stores [23], and is widely used in the research literature. We extended it so that the YCSB client emulator sends requests through a front-end load-balancer to one of application servers where the queries are executed within a web servlet, instead of sending them directly to a backend database. The backend database system is MySQL[3] 5.5.41. We work with a database size of 5GB ensuring that not all data fits into the AS caches but small enough so that our database backend does not become the bottleneck. YCSB comprises a set of workloads that emulate various data access behaviors. For our experiments we chose the *read latest workload* with a combination of scan and insert requests. Each scan request returns the first five tuples with key values less than or equal the included key parameter. Whenever a new item is inserted, a new request type is created accordingly with a key parameter equal the inserted key. Such workload exhibits request overlap and, at the same time, since the last created request type is the most popular request and the last inserted key is the most popular object, it also has a dynamic behavior.

[3] https://www.mysql.com/

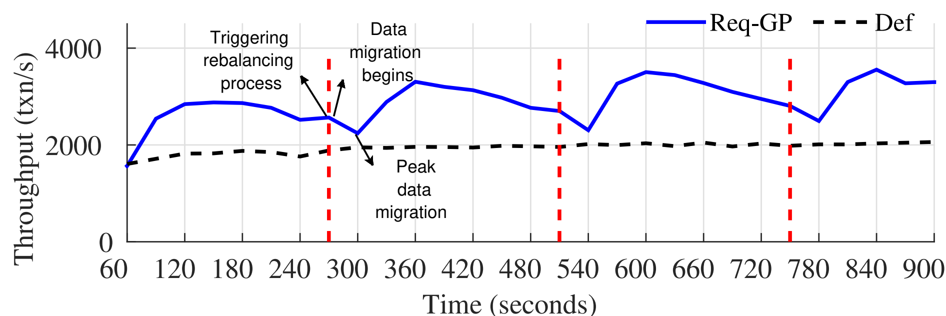
RUBiS: RUBiS[4] is a benchmark modeled after eBay to test application server performance, and is considerably more complex than YCSB. It has several tables representing buyers, sellers, items to be sold or auctioned in different categories and regions, bids, etc. Our test database has roughly 7GB of data. We use PostgreSQL 9.2.4 as backend. There are many request types such as browsing different regions and categories, items, bids, and the user profile. Users can also insert and purchase items. These different request types overlap in the objects they access. However, by default, RUBiS does not have requests with object drift as the request to search items always returns the same objects based on best match. Therefore, we add two new request types to search for items similar to those we found in eBay. The first sorts items based on their insertion time with newly listed items appearing first. This is a request with moderate drift rate. The second sorts items based on their expiration date with items with smaller expiration time appearing first. Furthermore, we force a large set of items to expire throughout the experiment leading to a request type with large drift rate. Furthermore, whenever the client emulator submits a search request for items, it chooses randomly one of the now three sorting orders (BestMatch, StartTimeNewest or EndingTimeSoonest). All three return 50 objects. There are a fair amount of smaller request types that return less objects.

3.3.2 Understanding Partitioning Behavior

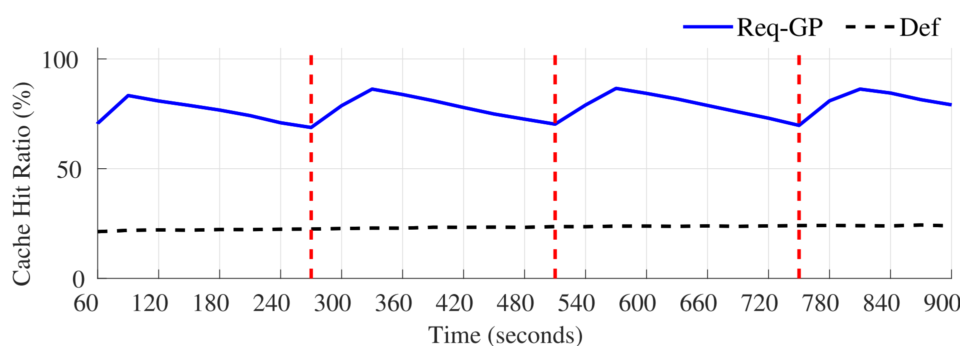
In our first experiment, we have performed a detailed runtime analysis for one of our partitioning algorithms, Req-GP, using the RUBiS workload to see the overall behavior of our system before, during and after reconfiguration. For the experiment, we have set the log window size to 60 seconds. We will later analyze this parameter in more detail. The threshold to trigger the partitioning is 10% (i.e., when the local hit ratio has degraded by 10% compared to the maximum local hit ratio since the last reconfiguration). We do not yet perform the optimizations described in Sections 3.2.4 and 3.2.5 but assign the new partitions randomly to servers and do not prune any objects or requests.

Figure 3.5 shows (a) throughput, (b) local cache hit ratio, and (c) the number of objects migrated after reconfiguration over time. The graphs start at 60 seconds, the time

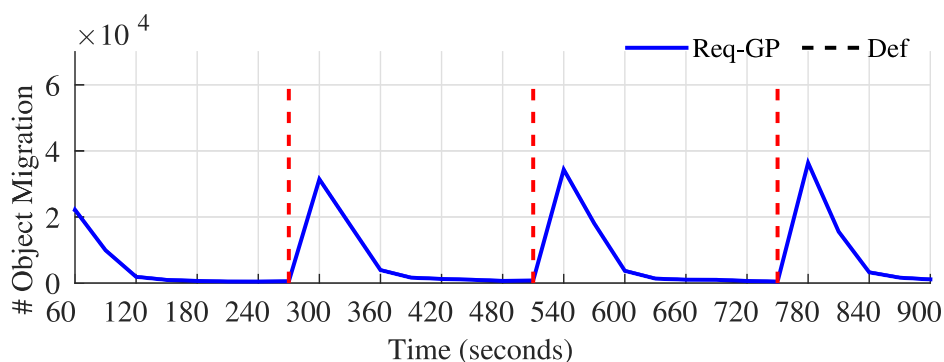
[4] rubis.ow2.org



(a) Throughput



(b) Cache hit ratio



(c) Data migration

Figure 3.5: Req-GP algorithm performance

the first partitioning takes place and end at 900 seconds of execution. The black dotted line shows the performance of standard round-robin load balancing without reconfiguration. Note that round-robin can also access data in remote caches. We refer to this default set-up as **Def**. We can see that Req-GP significantly outperforms simple Def at

all times. After the first reconfiguration at 60 seconds, throughput goes up, and then slowly degrades as the policy generated at 60 seconds does not serve well anymore the dynamic workload. We can see that the throughput is highly correlated with the local cache hit ratio. At 270 seconds (first vertical red line), the local hit ratio has decreased by 10% compared to its maximum value, and the Analyser generates and installs the new Request- and ObjectPolicies[5]. After they are installed, throughput actually first decreases somewhat as objects need to be migrated first (see Figure 3.5c), which puts some additional burden on the system and reaches its low at around 300 seconds. Right after that point and when most of the objects in the ObjectPolicy have been migrated, cache hit and throughput again increase. The maximum cache hit ratio is reached at around 360 seconds before it decreases again due to workload changes, and the cycle repeats.

Note that overall hit ratio, i.e., local and remote together, stays above 75% throughout the experiment as our architecture is able to use the entire cache capacity of all caches. As new objects are created they reside first in the cache of the server that created the object, and then will be migrated by our policy to shift the cache hit ratio towards local hits.

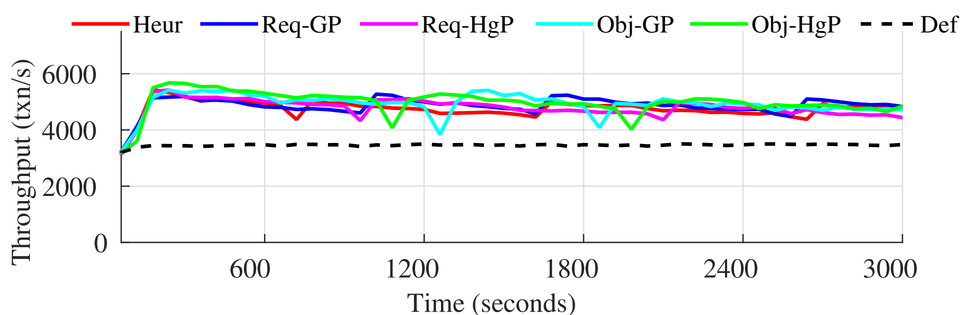
Thus, our system is able to use the entire cache capacity while, at the same time, keep the local hit ratio high, and this in a dynamic environment.

3.3.3 Algorithm Comparison

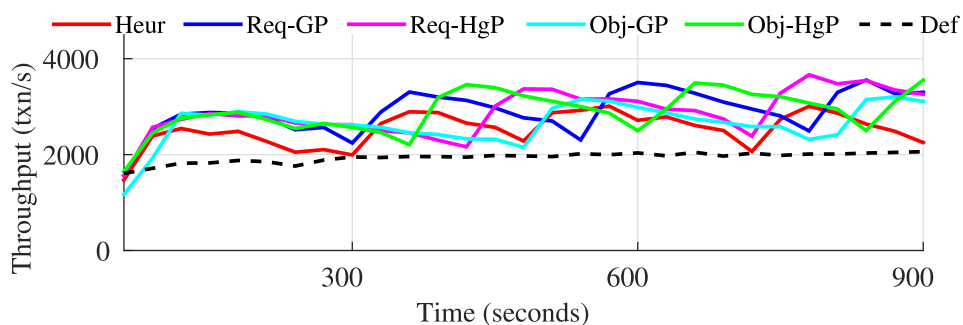
In this section, we have a closer look at the different partitioning strategies that we explore. This time, we use both YCSB and RUBiS, have again a 60 seconds log window size and a threshold for repartitioning of 10%. As deterioration was slower in YCSB, we ran it for 50 minutes. Figure 3.6 shows the throughput for (a) YCSB and (b) RUBiS over time, as well as (c) the overall throughput over the entire execution time for all algorithms for both YCSB and RUBiS.

All algorithms follow the overall behavior of Req-GP that we have observed in the previous experiment. Performance is always better than Def. After reconfiguration, per-

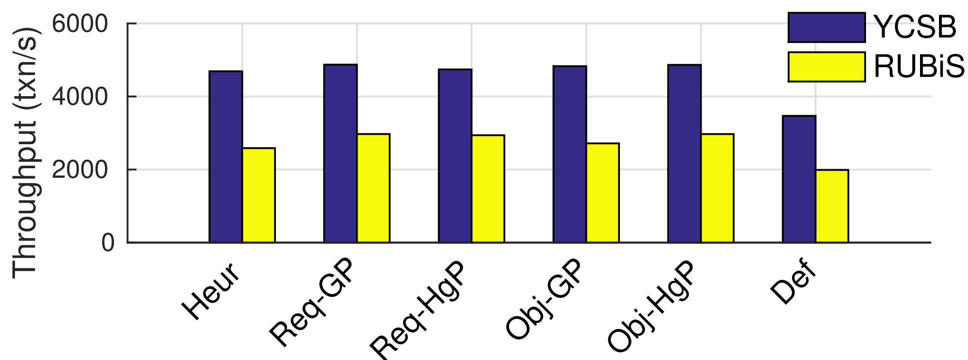
[5] As we will see later this process only takes a few seconds



(a) YCSB throughput



(b) RUBiS throughput



(c) Average throughput

Figure 3.6: Partitioning Strategies

formance first decreases until migration is completed and then increases quickly and stays high for a while. It then goes down again because of workload changes.

For YCSB, all algorithms provide similar performance. This is due to the simplicity

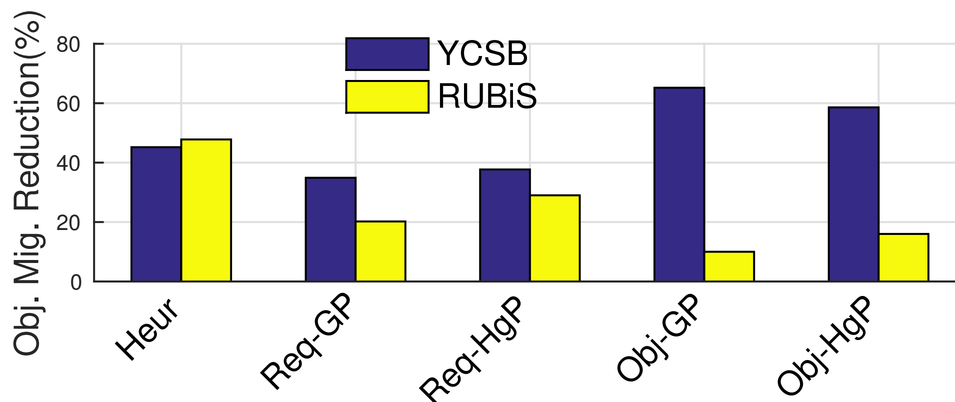
of the YCSB workload that consists of a single scan request targeting one table. Obj-GP and Obj-HgP both have worse throughput at peak migration time than the others because they do not per se guarantee an equal distribution of objects across all partitions as we weight our graph with load numbers. As a result, for YCSB, object distribution becomes very skewed: two of the caches have only 3% of the objects (which are very highly loaded) while the other two share the rest. If these latter objects are not assigned to the same server after reconfiguration, a huge data migration is triggered. Note also that this could potentially be a severe problem if cache sizes were more restricted. Note that the algorithms that partition requests first do not have this problem as we control the maximum number of objects that are assigned to any server after the partitioning of the requests has taken place. However, overall the performance differences do not look significant, and even our simple heuristic works quite well.

In case of RUBiS, re-balancing occurs at different times for the different algorithms. This is due to mainly two reasons. First, generating and partitioning the graph takes longer for some strategies. This is particularly the case for Obj-GP as we will see in Section 3.3.5. Second, in our implementation, the analyzer checks the degradation threshold only every 60 seconds. Thus, a small difference in the cache hit value of two algorithms might lead to one being triggered much earlier than the other. These differences in timing are therefore probably an artifact of our implementation. In fact, Figure 3.6c shows that all algorithms have nearly the same throughput for RUBiS with the exception of Obj-GP, which we will analyze later, and the simple heuristic. Given the higher complexity of the workload, a simple heuristic is not good enough anymore. In our case, the heuristic, for instance, does not take edge weight into consideration.

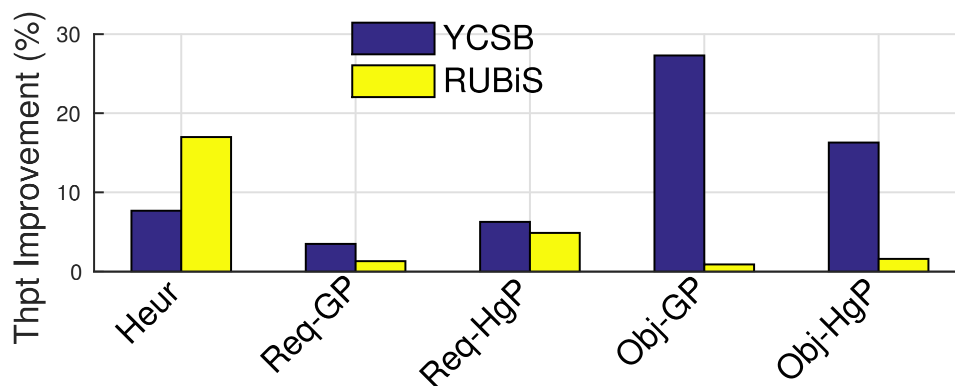
Overall, we can say that all graph representations provide good potential to properly present the workload of the system. The important thing is that both requests and their loads, and objects and their load and number, are properly taken into consideration when describing the workload. The object-based partitioning algorithms do not guarantee an equal distribution of objects in all cases (although they provided it in the case of RUBiS).

3.3.4 Assigning Partitions to Servers

In this experiment, we evaluate the impact of ensuring that partitions get assigned to servers in a way as to minimize the number of objects that need to be migrated. For that, we compare the throughput results of the experiment of the previous section with the throughput achieved when we assign partitions according to the previous cache content of the servers (see Section 3.2.4).



(a) Data migration reduction



(b) Throughput improvement

Figure 3.7: Smart Assignment of Partitions to Servers

Figure 3.7a shows for YCSB and RUBiS for all partitioning strategies, how much

we were able to reduce the amount of objects migrated with our cache-aware assignment of partitions to servers. For the Heuristic, we had to migrate roughly 45% less objects for both benchmarks. This is due to the simplicity of the partitioning algorithm, where the same requests tend to end up in the same partition and still access a fair amount of objects that they have accessed in the past. Thus, there is a lot of overlap between the new object partitions and the content of the individual caches, and our cache-aware assignment leads to less data migration.

For the other partitioning algorithms, the gain for RUBiS is fairly small. These algorithms are more sensitive to graph changes since they do not only consider edge connectivity, but also edge weights. In addition, they use several optimizations to enhance the quality of the produced partitions. Thus, they produce partitions with different content every time, and therefore, there is less overlap with the current object distribution.

The results are better for YCSB because of the simplicity of the benchmark. The graph remains very close from iteration to iteration. Thus, the algorithms tend to build very similar partitions every time and a cache-aware assignment leads to a significant reduction in object migration. For the object-based partitioning, a further aspect is that, as we already mentioned, they produce skewed object distributions for YCSB as we partition based on object access frequencies (i.e., load), and not according to total number of objects per partition. In this particular case, one server has nearly 60% and another nearly 40% of all data. If we guarantee that we assign these object partitions again to the servers that already have the data, very few data has to be migrated.

Figure 3.7b shows the improvement in throughput during the migration phase that is achieved by the cache-aware partition assignment. We can see that the impact on throughput is somewhat limited most of the time but quite significant for the Heuristic and for the object-based partitioning algorithms for YCSB that now avoid the deep throughput dip that was observed in Figure 3.6a.

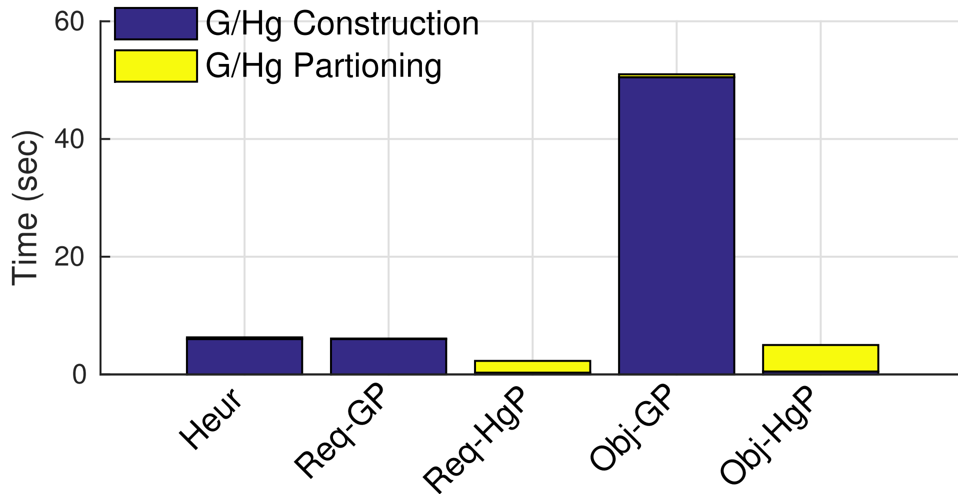
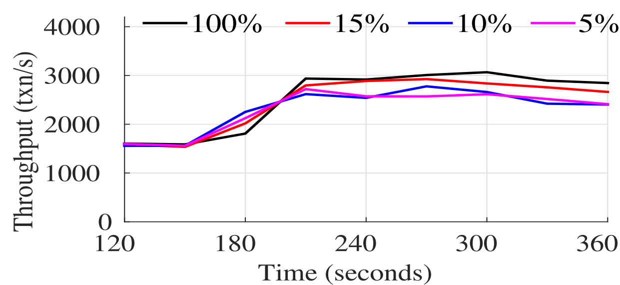


Figure 3.8: Graph Construction and Partitioning Time

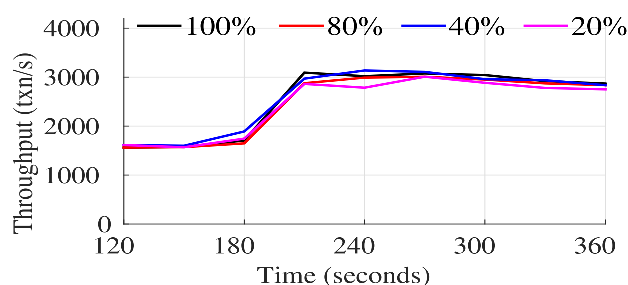
3.3.5 Meta-data Pruning

One of the challenges for dynamic reconfiguration is to generate the caching policies in a relatively short time. Figure 3.8 shows the processing time considering both graph construction and graph partitioning for all of our algorithms when considering 60 seconds of log information. All except one algorithm require less than 7 seconds. The hyper-graphs require less time for graph creation (as they create less edges) but more time for partitioning. In fact, partitioning time is negligible for all simple graphs. Req-HgP is the fastest of all. In contrast, Obj-GP takes considerably longer. The reason is that we consider all requests, including those with high drift rate. However, these requests generate many edges as they access many objects over the observation interval. Thus, the construction of the graph takes very long. For instance, the request-based graph has 15000 request vertices and 120000 edges while the object-based graph has 150000 object vertices and 2 Mio. edges between them. Heur and Req-GP behave roughly the same because the request-based graph is constructed the same way in both cases and partitioning time is negligible.

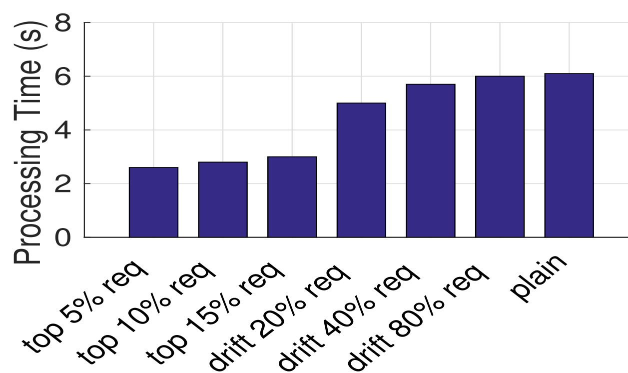
The question now is how pruning requests and objects can increase performance both



(a) Top k% req. thpt.



(b) Drift thpt.

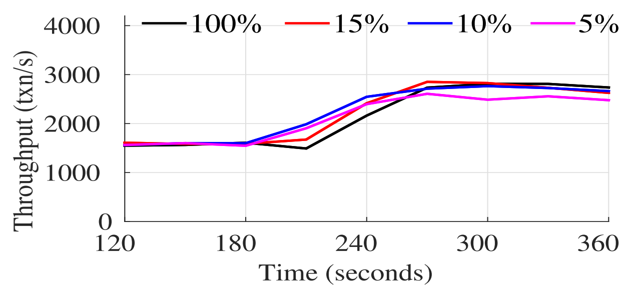


(c) Processing time

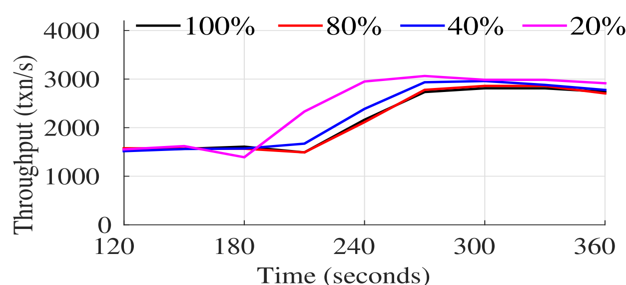
Figure 3.9: RUBiS and Req-GP: pruning meta-information

in terms of partitioning time as well as data migration. We run this set of experiments with Req-GP, as a representative where graph processing is fast, and Obj-GP, our outlier.

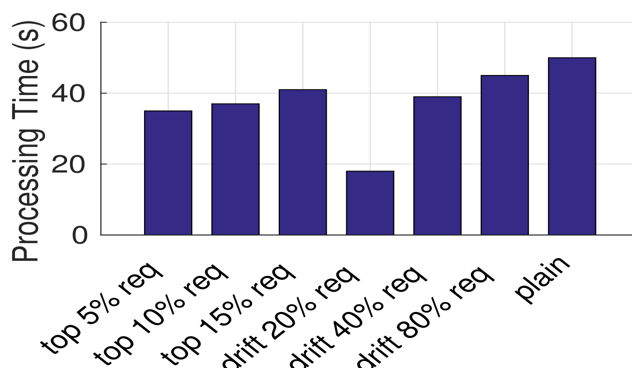
We first analyze pruning unpopular requests and then pruning requests with high drift rate. Figures 3.9a and 3.10a show throughput for Req-GP and Obj-GP, respectively, con-



(a) Top k% req. thpt.



(b) Drift thpt.



(c) Processing time

Figure 3.10: RUBiS and Obj-GP: pruning meta information

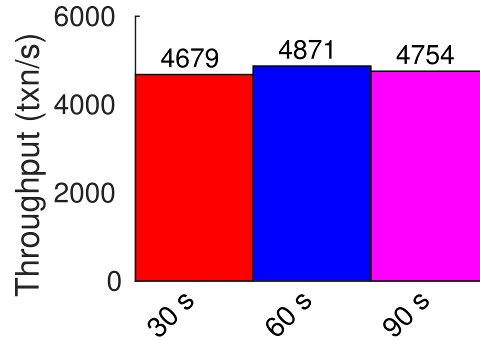
sidering 100% of the requests, or only the 5%, 10% and 15% highest weighted requests. Additionally, the first three bars in Figures 3.9c and 3.10c show the time to generate the partitions. For Req-GP, considering less requests, at time of reconfiguration (180 second), throughput increases quicker as less objects are migrated, but after that throughput is significantly lower. Pruning 90% or more of the requests leads to a throughput loss

of around 15% which is mainly due to a lower local hit ratio as many requests are not considered, and thus are not sent to the right application server. At the same time, Figure 3.9c shows that Req-GP runs in a bit less than half of the time (2.5 seconds vs 6 seconds). In contrast, and as shown in Figure 3.10c, the reduction in Obj-GP run-time from 50 seconds to around 35 seconds allows reconfiguration to kick in earlier and thus, throughput increases faster than if all requests need to be considered. Also, the number of objects to be migrated is significantly less leading to nearly the same performance despite the pruning.

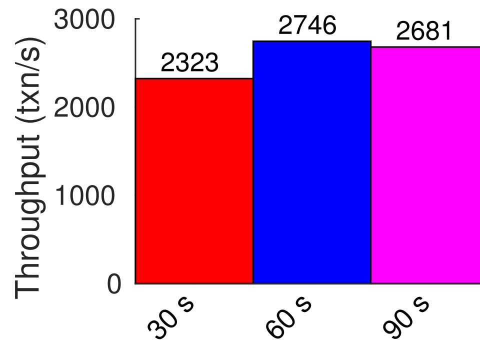
In a second experiment, we analyze the impact of eliminating requests that have high drift rate. Again, we run both Req-GP and Obj-GP against RUBiS considering all requests or only the 20%, 40%, and 80% of requests with lowest request drift. Figures 3.9b and 3.10b show throughput for Req-GP and Obj-GP. Again, we refer to Figures 3.9c and 3.10c for the time to generate the partitions with these drift rates. For Req-GP, eliminating high drift requests does not have a noticeable impact on the throughput but slightly reduces the processing time. Considering high drift-rate requests does neither harm nor benefit the performance. In any case, their objects are likely to not be in the right cache. In case of Obj-GP, the outcome of eliminating these requests is better as graph partitioning is much faster, in particular if we only consider the 20% of requests with lowest drift rate. Thus, reconfiguration can kick in faster and benefits are quicker observable.

3.3.6 Log Window Size Analysis

So far, we only presented results using 60 seconds of logging information. We also experimented with different intervals, and found the performance to be quite similar. For instance, Figure 3.11 shows the throughput with Req-GP for YCSB and RUBiS log window sizes of 30, 60, and 90 seconds. 60 seconds achieves slightly better performance than 30 and 90 seconds. This is probably due to the trade off between two contradicting factors. With a large window size, some of the workload data might be outdated; with a short window size we might not capture all necessary information. In our current system, the interval needs to be set manually. In future work it would be interesting to find the



(a) YCSB- Avg. thpt



(b) RUBiS -Avg. thpt

Figure 3.11: Impact of time window size

sweet spot dynamically and workload dependent.

3.3.7 Result Highlights

At first view, all approaches appear to behave sufficiently well. Thus, we believe a graph-partitioning approach is a feasible and promising solution to our caching challenge.

A closer view reveals several interesting details. First, the behavior of the simple YCSB differs significantly from the more complex RUBiS. In particular, the partitions from one iteration to the next are quite similar, thus, our smart partitioning assignment has significant benefits. However, RUBiS reveals that as soon as the workload and data design becomes slightly more complex, the generated graphs and resulting partitions

differ a lot from one iteration to the next. Thus, graph-partitioning is quite disruptive and not an incremental approach. Therefore, having a lazy and smooth data migration solution is crucial.

In the same realm, with a simple heuristic approach, while being potentially more incremental and thus, less disruptive at time of reconfiguration, it is challenging to come up with partitions that have equal performance compared to when using optimized graph-partitioning algorithms, in particular when data design and access patterns are complex.

The object-based graph, as being proposed previously in the literature, has difficulties when requests have a large drift. The size of the graph can quickly become unmanagable. Request graphs tend to be smaller as the number of different requests is potentially considerably smaller than the number of different objects. While the size of an object-graph can be managed through the request pruning mechanisms that we propose, it appears that avoiding the problem all together by using request-based graphs appears appealing. Furthermore, in contrast to request graphs, controlling the number of objects assigned to each partition when considering object load is not feasible, which might lead to poor cache size utilization.

Pruning can significantly improve the performance in some cases and appears to have little negative effect in the other cases. Thus, pruning is an effective mechanism to handle the problem space.

4

Consistency and Space Aware Cache Replication

In the previous chapter, we showed how carefully partitioning requests and objects across application servers and their caches ensures a high local cache hit and an overall low cache miss rate. How good such partitioning approaches can be in terms of high local hit rates depends heavily on the respective workload. As current enterprise workloads tend to be complex, user requests typically access several objects, different requests access overlapping object sets, object access is highly skewed, and object popularity changes over time, it is not always possible to perfectly partition the objects such that each request finds all objects it accesses locally. A widely used technique to overcome such workload complexity is object replication. In its simplest form, an object can be replicated to all caches where it is locally needed, and we avoid having remote reads all together. However, object replication comes with two major challenges: the first one is maintaining replica consistency under update workloads that can, such as in strong consistency, significantly increase response time for write operations. And the second challenge is that object replication consumes the limited cache space which potentially results in a larger number of cache misses for other objects leading to expensive back-end access.

In this chapter, we present a replication solution that replicates objects to specific caches only if there are real performance benefits. We assume a partitioning approach

has distributed objects across partitions and the load-balancer has some policies to send requests to partitions. Ideally, these policies send requests to partitions that have the necessary objects local most of the time. For instance, any of the distributed approaches presented in the previous chapter can be used. On top of this, we perform replication independently from the object distribution approach. That is, we add replicas for some of the objects, namely whenever the performance gain for reads outperforms the penalty of data consistency and reduced cache space. Furthermore, we show how the proposed replication solution is dynamic, detecting workload changes as they occur and reacting to them in an eager manner without service interruption. Finally, we present extensive performance tests using the RUBiS and YCSB benchmarks showing how our replication solution improves performance over a distributed cache and outperforms an existing replication solution.

4.1 Replication Challenges

Theoretically, a partitioning algorithm should allow every request find all objects in the local cache. However, our workload analysis as well as previous studies [91, 61] have shown that enterprise applications are complex, and partitioning solutions are always best-effort: there will always be the need to retrieve some objects from remote caches. In order to avoid remote reads, we can replicate objects. But this will increase write costs due to consistency requirements, and might fire back as our overall capacity of caching different objects is reduced. In the following, we discuss these two challenges in addition to the dynamic workload challenge in more detail.

4.1.1 Data Consistency

When an object is updated its physical copies need to be updated. There exist a plethora of data consistency protocols, and several attempts in the research literature to categorize them [56, 122, 115, 114, 8, 9]. *Weak consistency algorithms* execute the operation on a single object copy and then confirm success to the user. Propagation of the update to other replicas occurs in the background. In this case, the response time observed by the user is usually not significantly affected by the number of copies. *Strong consistency algorithms* perform the update on more than one copy before a confirmation is sent to the user, typically on a quorum or on all copies. Some implementations only ensure that the request is received at the remote cache while others wait until the update is actually executed at all caches. All this can have a significant impact on the response time. Again, many different algorithms and implementations exist which can result in very different response times.

As an illustrative example of the performance differences between algorithms, we run a simple experiment on the cooperative cache with EHCache which we extend with replication. We use the extended servlet version of Yahoo! Cloud Service Benchmark (YCSB) [23] that was discussed in Section 3.3.1 and submit a YCSB workload with 50% update requests, each updating 10 records, and 50% scan requests, each reading 100 objects. We use a weak consistency and a strong consistency protocol, the latter

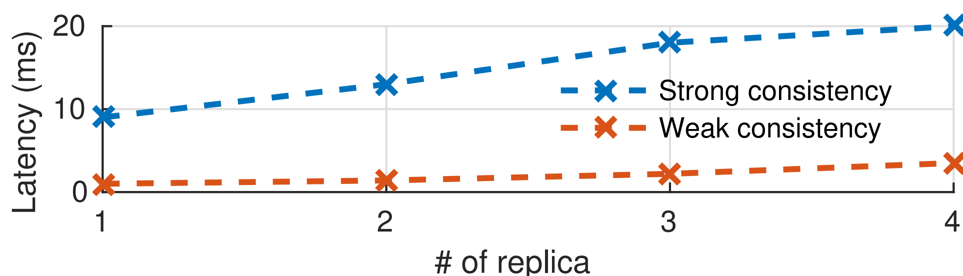


Figure 4.1: Impact of number of replicas on update latency

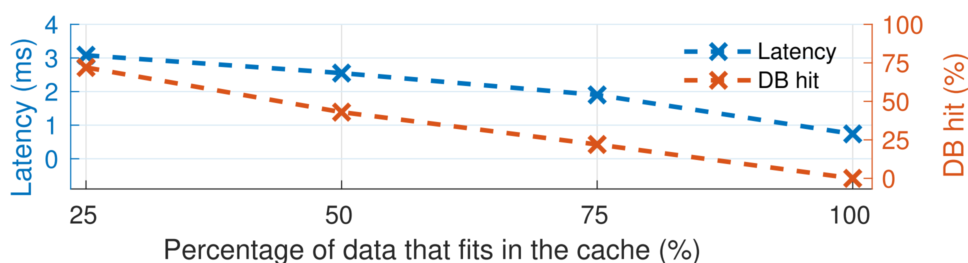


Figure 4.2: Impact of limited cache space on latency

updating all replicas before returning to the user.

Fig. 4.1 shows the average update latency per object for different number of replicas. As expected, there are large differences in the update costs and in the dependency on the number of replicas. Clearly, performance differences do not depend only on the particular algorithm but also how it is exactly implemented and engineered.

Showing these extra costs, it becomes clear that replication can induce a considerable or very little overhead on write operations. Sometimes, the negative effects might be considerably higher than the gains for read operations.

4.1.2 Limited Cache Space

The cache capacity of all cache instances of a cooperative cache could still be too small to hold the most frequently accessed objects. In such a case, replicating some objects will aggravate this problem as even less different objects will fit into the caches, leading

to overall lower cache hit (local or remote) and more database access. Thus, while access to replicated objects will now be faster, access to objects that can no more be kept in the cache will be a lot more expensive.

To show the impact of limited cache sizes, we set up a cooperative cache with two cache instances and run the extended YCSB workload composed of 100% scan requests where each request scans 100 objects (uniform distribution). We set up the caches so that the aggregated cache size for both caches holds 25%, 50%, 75%, or 100% of all accessed data (randomly partitioned). When a data item is accessed that is not in one of the two caches, the data item has to be retrieved from the backend database. Fig 4.2 shows the response time of the requests on the left y-axis and the percentage of database accesses on the right y-axis. With increasing cache capacity latency decreases. One can see that this is directly related to the decreased database access.

In summary, the trade-offs between replicating some objects causing other objects to be removed from the cache vs. the capacity to hold a larger number of different objects has to be well analyzed.

4.1.3 Dynamic Workload

As discussed earlier, workload can quickly change. One particular change that can have a significant impact on a replication solution is a change in the read/write ratio overall and of individual objects. This can happen due to several factors. For example, price drops for certain items on an e-commerce website may lead to a spike in their sales. Or in an auction-bidding system such as eBay, bidding on items might increase just shortly before their expiration. Furthermore, an online ticketing system might experience a spike in ticket sale when the corresponding event approaches. These update spikes on an object change the read/write ratio and may diminish the replication benefit or even worse, lead to higher response time than without replication. Thus, our replication solution needs a good mechanism to detect such read/write changes, and then react to them without causing too much reconfiguration overhead.

4.2 AdaptCache Replication Extension

In order for AdaptCache to support dynamic object replication, we have empowered the cooperative cache framework with the necessary tools to be able to carry dynamic object replication. In particular, we have extended the cooperative cache to support consistency in case of replication by implementing a strong consistency algorithm that updates all cached objects and the database object before returning to the client. However, our system is not dependent on this particular algorithm as we see further below.

In addition we have extended the parameter collection process, described in Section 3.2.1, to support dynamic object replication. In particular we have extended the coarse- and fine- grained tracking phases to collect replication-related parameters. Also, we have added an extra feature for the AdaptCache to extract cache operation costs (latency) that are necessary for our solution.

4.2.1 Extending the Parameter Collection

In Section 3.2.1, we showed how the Analyser manages the parameter collection process through coarse- and fine-grained tracking in order to detect workload changes and collect necessary parameters for conducting data partitioning. Here, we will show how we extend these two phases to support dynamic object replication.

Extending Coarse-grained Tracking

Recall that in the coarse-grained tracking phase, the Analyser tracks only a high level workload information that is required to decide if data reconfiguration is necessary. In addition to tracking the local cache hit ratio that is used to decide if a new repartitioning is necessary, we extend the Analyser to keep track of the the total number of read and write operations to determine the read/write ratio. A read/write ratio is a strong indicator that can be used to detect changes on the write request frequency submitted to the system. A change of such frequency may impact the current replication deployment and thus may trigger a replication adjustment. To this end, the Analyser maintains two counters, one

Table 4.1: Parameters for object replication

n_p	number of partitions
$o.n_{rp_i}$	number of reads to object o issued by partition p_i
$o.n_r$	$\sum_{i=1}^{n_p} o.n_{rp_i}$
$o.RP$	set of partitions with at least one read operation to o i.e., $p_i \in o.RP$ iff $o.n_{rp_i} > 0$
$o.n_{wp_i}$	number of writes to o issued by p_i
$o.n_w$	$\sum_{i=1}^{n_p} n_{wp_i}$
$o.WP$	set of partitions with at least on write operation to o i.e., $p_i \in o.WP$ iff $o.n_{wp_i} > 0$

for the number of read-object operations and the other for write-object operations. For simplicity, whenever an object log message of type *get* arrives during the coarse-grained phase, the read counter is increased wheres the write counter is increased if the object message has a put type. Similar to monitoring the local hit ratio, the Analyser keeps track of the read/write ratio to detect workload changes.

Similar to Section 3.2.1, the Analyser periodically checks first if there is a significant change in the local cache hit ratio. If this is the case, the Analyser determines a new partitioning solution followed by determining a new replica assignment. If the local cache hit ratio has not changed, there is no need to repartition. But if the read/write ratio has significantly changed, then the Analyser will still revise the replica assignment as the write ratio can have a considerable impact on the benefits of replication.

Extending Fine-grained Tracking

The fine-grained tracking phase aims to collect information about requests and objects that are necessary to perform partitioning and replication. In the previous chapter, we showed the collected information to perform partitioning. Similarly, in order to assign replicas, the Analyser generates statistics for each accessed object and that for a predefined observation interval. These metrics are listed in Table 4.1. We refer to a partition p_i

as the union of all client read-only and update requests that are assigned to server i and all the objects that are stored in the server's local cache. There are a total of n_p partitions. For each object o that was accessed during the interval of observation, we denote with $o.RP$ ($o.WP$) the set of all partitions that have at least one request that reads (writes) o. For each $p_i \in o.RP(o.WP)$, the Analyser keeps track of the number of read (write) operations that originate from p_i denoted as $o.n_{rp_i}$ ($o.n_{wp_j}$). Also, the Analyser keeps track of the total number of reads and writes that originate from all partitions to object o denoted as $o.n_r$ and $o.n_w$ respectively. The operation here refers to the individual cache operation that is spawned by a request type. A scan request, for example, that targets 10 objects and is executed at partition p_i will spawn 10 read operations all of which originate from p_i.

Note that while both partitioning and replication solutions use the same workload meta-data to collect input parameters, the collected statistics are mostly different. A very few parameters are shared across both solutions; such as the number of partitions and the object's total number of reads. The rest are different. Specifically, the replication solution does not need information about individual read requests such as their types or access frequencies. This is because the replication solution relies on the underlying data partitioning solution to manage request distribution; thus, it does not need information about individual read requests. It does need, however, information about the number of read and write requests issued to objects as well as from which partitions. This is because the presented replication approach is an object-based solution that manages object replica locations across caches based on various object-based statistics. These statistics are indirectly derived from requests as they access objects. That is, for replication purposes, we only need the information provided in the object logs.

4.2.2 Extracting Operation Costs

We mentioned before that replication leads to faster reads but slows down writes. To understand the trade-off we need to quantify the performance. To this end, we leverage response time as our performance metric because it reflects well user experience, and because it can be automatically and transparently extracted. Since the overall object re-

Table 4.2: Cost (Latency)

c_{rl}	cost of read local
c_{rrem}	cost of read remote
c_{rdb}	cost of read db
$c_{wl}(k)$	cost of writing k replicas (one write local)
$c_{wrem}(k)$	cost of writing k replicas (only remote writes)
c_{wdb}	cost of only writing to db

sponse time depends on individual cache operation costs or response times, we extract all object-related operation costs from our running system. Table 4.2 lists the different costs that we measure in our system. For reads, c_{rl} indicates the time to retrieve an object from the local cache, c_{rrem} is the retrieval time from a remote cache and c_{rdb} is the time to retrieve an object from the database.

Write costs are a bit more complicated as they also depend on the number of replicas, especially for strong consistency protocols. Therefore, $c_{wl}(k)$ depicts the cost for a write operation with k replicas where one of the replicas resides in the local cache, while $c_{wrem}(k)$ depicts the cost where all replicas are in remote caches. We can expect $c_{wl}(k)$ to be lower than $c_{wrem}(k)$. However, with increasing k we can expect this benefit to be less and less. Finally, we also extract the write cost c_{wdb} when the object is in no cache and only the database has to be updated. c_{wdb} will always be lower than when the object is cached as the database must always be updated.

4.2.3 Independence from Distributed Solution

So far, we have discussed the replication solution as an extension to our distribution solution discussed in Chapter 3. But we can also use it independently as the replication solution is decoupled from the distributed solution. To do so, the targeted system has

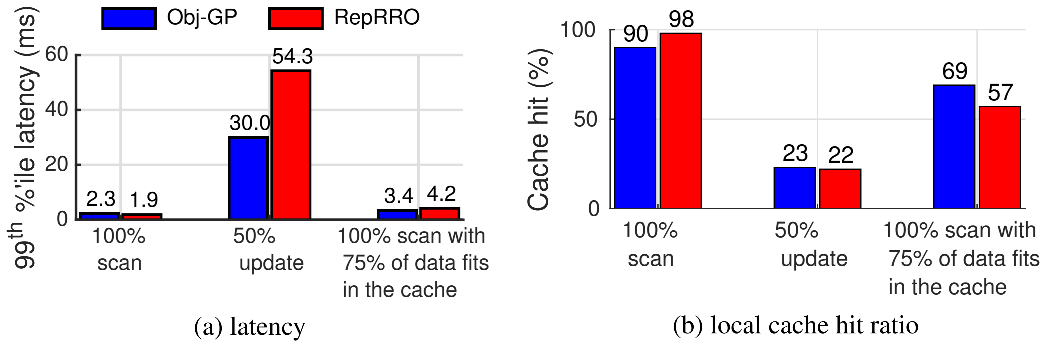
to have certain requirements. First, the data has to be well distributed across sites in a way that groups a set of data that is accessed together in the same partition. Second, a load-balancer has to be able to distribute incoming requests across these partitions such that most requests can find most of their data in the partition where they are executed and minimize the number of remote data access.

Once the data has been well distributed, object replication parameters and operation costs, similar to those in Tables 4.1 and 4.2, can be extracted, even differently, and used by the replication solution that is totally independent from the underlying distributing solution. The independent nature of the replication solution manifests itself in several ways. First, it does not need to know how the underlying partitioning solution works. Second, it does not need to know the implementation details of read/write operations or the used consistency approach. Third, it does not need to know information about request types, their access frequencies or sites where they are executed. Such independence allows the replication solution to gracefully work on top of other data distribution solutions. In Chapter 7 of the Future Work, we will discuss the feasibility of adding our independent replication solution on top of well-known systems and partitioning solutions.

4.3 Basic Replication

In this section we discuss a simple replication solution that serves as our baseline. As our solution assumes that the already deployed partitioning solution assigns each object to a single partition and distributes requests across server, replication is now relevant for an object o that is assigned to partition p_i but there is a remote partition p_j that needs to execute a read operation on o. We refer to these objects as remote read objects, or RRO.

With this, the basic solution, also denoted at $RepRRO$, extends the base distribution solution by adding copies so there are no more RROs in the system. More precisely, a copy of an object o is added to a partition p_j to which o was not originally assigned to, if and only if there is a read operation on o originating from p_j. As a result, all originally remote reads are transformed to local reads.



(a) latency

(b) local cache hit ratio

Figure 4.3: Partitioning vs Basic replication performance

$RepRRO$ focuses on making reads fast as they will be on local caches. However, it has only little consideration for the additional overhead in case of updates. Furthermore, it does not take limited cache capacity into consideration.

To better understand the limitations, we compare this basic replication solution with one of the non-replicated distribution approaches presented in Section 3.2.2, specifically the object-based graph partitioning, which we refer to as *Obj-GP*, that achieves fairly high but not perfect local cache hit ratio. We use the YCSB benchmark and four caches to analyze three different workloads. Workload I has only scan requests each accessing 100 objects. Workload II has 50% update requests each reading and then updating five objects, and 50% scan requests (as before). We use a strong consistency algorithm as described in Section 4.2. In both workloads, all objects fit in the aggregated cache. Workload III has 100% scan requests as before but the aggregated cache space can hold only 75% of the total number of accessed objects.

Figure 4.3a shows the 99th percentile latency per operation and Figure 4.3b the local cache hit ratio. Lowest latencies are with workload 1. Replication has a higher local cache hit ratio, and thus, lower latencies than *Obj-GP*. In update workload II, both approaches perform worse than the read-only workload due to the high cost of writing to the backend database. Replication is actually worse than *Obj-GP* due to the extra time required for updating the additional copies. The gain for the read operations is overall less than the extra overhead for writes. For workload III with a limited cache space, replication achieves again worse performance than *Obj-GP*, this time because the repli-

cas consume the limited cache space and, thus, the overall cache hit (local and remote together) is smaller, triggering a higher DB access.

Clearly, such a simple basic replication approach is not sufficient to balance the trade-offs of replication. In the next two sections, we will show how to tackle these challenges.

4.4 Managing Update Overhead

In this section we look at the trade-offs between the benefits of reads and the overhead of writes, but we ignore limited cache sizes. We start again with the partitioning only solution. We are only interested in objects that have remote access, that is, an object o assigned to partition p_i and there is a partition p_j that has an operation on o (i.e. $|o.RP \cup o.WP| > 1$). As a first step, we check for each of these objects whether it is a read-only object ($o.WP = \emptyset$). If this is the case, we simply follow the basic replication approach by replicating the object to each $p_i \in o.RP$ that does not store o originally. As there is no overhead for writes, it makes sense to replicate the object to all partitions that access it.

If an object o is (also) updated we have to be more careful. We want to find the set of partitions $o.PC \subseteq o.RP \cup o.WP$ such that replicating object o to all partitions in $o.PC$ maximizes the gain in performance for o.

4.4.1 Calculating Execution Costs

For that, we first must be able to quantify performance. We do this by calculating for a potential configuration $o.PC$, the expected total overall execution time, denoted $TT(o.PC)$. The calculation takes all read and write operations on o that were observed during the observation interval, and sums up their expected execution time should o be replicated to all partitions in $o.PC$. The configuration with the lowest overall time has the lowest cost, and thus, is considered the best configuration.

We discussed in the previous section the parameters that we need to calculate this

cost: we need to know how many read and write operations to o were submitted at each partition (n_{rp_i} and n_{wp_i}), the costs for local and remote cache reads (c_{rl} and c_{rrem}), and the costs for write, depending on the number of replicas and whether the submitting partition has a local copy or not ($c_{wl}(k)$ and $c_{wrem(k)}$).

Using these parameters, we calculate the total expected execution time when replicating o to all partitions in $o.PC$ as the sum of expected read time $RT(o.PC)$ and write time $WT(o.PC)$:

$$TT(o.PC) = RT(o.PC) + WT(o.PC) \tag{4.1}$$

The formulas for read and write costs are quite similar. For reads, let $o.n_{rl}(PC) = \sum_{p_i \in o.PC \cap o.RP} o.n_{rp_i}$ be the number of all read operations where there is a local replica in configuration $o.PC$, and let $o.n_{rrem}(PC) = \sum_{p_i \in (o.RP \setminus o.PC)} o.n_{rp_i}$ be the number of all read operations where there is no local replica. Then we have

$$RT(o.PC) = o.n_{rl}(PC) * c_{rl} + o.n_{rrem}(PC) * c_{rrem} \tag{4.2}$$

That is, all reads on partitions with replicas have local read cost, and all reads on partitions without replicas have remote read costs. We do not need to look at partitions with no read operations. The larger the number of reads n_{rp_i} and the more expensive a remote read is compared to a local one ($c_{rrem} - c_{rl}$), the lower will be the total time if o has a replica on p_i.

For writes, let $k = |o.PC|$ be the number of replicas, let $o.n_{wl}(PC) = \sum_{p_i \in o.PC \cap o.WP} o.n_{wp_i}$ be the number of all write operations where there is a local replica in configuration $o.PC$, and let $o.n_{wrem}(PC) = \sum_{p_i \in (o.WP \setminus o.PC)} o.n_{wp_i}$ be the number of all write operations where there is no local replica. We have

$$WT(o.PC) = o.n_{wl}(PC) * c_{wl}(k) + o.n_{wrem}(PC) * c_{wrem}(k) \tag{4.3}$$

Again, we do not need to consider partitions that do not write o, but we have to consider the number of replicas k. While the number of replicas negatively affects writes, for a particular partition the access cost with a local replica $c_{wl}(k)$ might actually be lower than a remote access with less overall replicas $c_{wrem}(k-1)$. This could be, e.g., the case with weak consistency. In this case, only one copy is updated first and update propagation

is performed in the background. Updating locally will likely be faster than updating a remote copy. For strong consistency algorithms, however, the number of replicas will have a much more negative affect. Given this complexity, it's much less clear how much the actual update overhead for writes really is.

Example Let's have a look at a simple example with two partitions p_1 and p_2, an object o originally assigned to p_1, read operations r_1 and r_2 accessing o and assigned to p_1 and p_2 respectively, and two write operations w_1 and w_2 on o assigned to p_1 and p_2 respectively. Assume first that $n_{rp_1} = 30$, $n_{rp_2} = 20$, $n_{wp_1} = n_{wp_2} = 2$, $c_{rrem} = 1$, $c_{rl} = 0.1, c_{wl}(1) = 8, c_{wrem}(1) = 10$, and $c_{wl}(2) = 12$.

Thus, we have three configurations with different total times as follows: $TT(o.\{p_1\}) = 3 + 20 + 16 + 20 = 59$, $TT(o.\{p_2\}) = 30 + 2 + 20 + 16 = 68$, and $TT(o.\{p_1, p_2\}) = 3 + 2 + 24 + 24 = 53$. Thus, configuration $o.\{p_1, p_2\}$, which replicates o to p_2 saves execution time. However, if we have more write operations, e.g., $n_{wp_1} = n_{wp_2} = 5$, then $TT(o.\{p_1\}) = 113$, $TT(o.\{p_2\}) = 122$, and $TT(o.\{p_1, p_2\}) = 125$ and we should not replicate.

Complexity Calculating the costs for all subsets of $o.RP \cup o.WP$ might appear to be very costly as there are $2^{|o.RP \cup o.WP|-1}$ possible subsets. However, if the partitioning solution did a good job, then $|o.RP \cup o.WP|$ will actually be a small number. However, we have mentioned that for the cooperative cache, it was actually very difficult to determine which objects a client write request would write; thus write requests are distributed randomly across partitions, and $|o.WP|$ is equal to the number of partitions in the system. Therefore, in the following, we will have a closer look at this assignment policy and see that in this particular case, complexity is not exponential. Instead we only have to look at n_p different configurations.

4.4.2 Random Write Distribution

In the partitioning solution of Chapter 3, the partitioning algorithm distributes write operations randomly across all nodes using round robin. With this, $|o.WP| = n_p$ is the num-

ber of partitions in the system and for any two $p_i, p_j \in n_p$, we can expect $n_{wp_i} = n_{wp_j}$. With $o.n_w = \sum_i o.n_{wp_i}$ being the total number of write operations and $k = |o.PC|$ the number of replicas, the write time becomes

$$WT(o.PC) = n_w(\frac{k}{n_p}c_{wl}(k) + (1 - \frac{k}{n_p})c_{wrem}(k)) \tag{4.4}$$

That is, $\frac{k}{n_p}$ of the n_w write operations can access a local copy while the rest has no local copy available. With this, the overall write overhead does not depend on where an object is replicated, but only how many replicas there are. The write costs for all configuration with k replicas are the same.

Two partitions Let's look at an example with two partitions p_1 and p_2, and o originally assigned to p_1, two read operations r_1 and r_2 assigned to p_1 and p_2 respectively, and writes equally distributed across both partitions. The question is now whether it's worth to replicate o to p_2 to make r_2's reads local. If we look at the two configurations we have

$$TT(o.\{p_1\}) = n_{rp_1}c_{rl} + n_{rp_2}c_{rrem} + n_w(1/2c_{wl}(1) + 1/2c_{wrem}(1))$$
$$TT(o.\{p_1, p_2\}) = (n_{rp_1} + n_{rp_2})c_{rl} + n_w c_{wl}(2)$$

Replicating to p_2 is better if $TT(o.\{p_1\}) > TT(o.\{p_1, p_2\})$ which can also be expressed as

$n_{rp_2}(c_{rrem} - c_{rl}) > n_w(c_{wl}(2) - (1/2c_{wl}(1) + 1/2c_{wrem}(1)))$

or

$\frac{n_{rp_2}}{n_w} > \frac{c_{wl}(2) - (1/2c_{wl}(1) + 1/2c_{wrem}(1))}{c_{rrem} - c_{rl}}$

That is, if the ratio of reads that turned local to the number of writes is larger than the ratio of the extra overhead of a write to the performance gain for a read, then replicating an object to p_2 will result in a better total time compared to not replicating it.

More than two partitions Interestingly, if write operations are equally distributed across all nodes, then we can significantly reduce the number of replica configurations

that we need to consider. More particular, the number of configurations for which we have to calculate TT to determine the lowest possible TT, is equal to the number of partitions n_p. More precisely, for each $k = 1...n_p$, we only need to calculate TT for one configuration with k replicas.

For $k = 1$, i.e., no replication, we assume that the distribution algorithm provided us with the best solution. Assume it assigned o to p_i. Then, there is no need to look at any other $o.PC = \{p_j\}$ where $j \neq i$. Thus, we only need to calculate TT of one configuration where $k = 1$.

From there, we sort the remaining p by descending number of read accesses n_{rp} into vector $(p_{j_2}, p_{j_3}, \dots)$. Partitions that do not have any read access will all be at the end of this list.

For $k = 2$, we calculate the total time for configuration $o.PC = \{p_i, p_{j_2}\}$. Among all the configurations with two replicas we can be sure that this one has the smallest TT. By adding the partition with the most read operations, we are able to reduce remote cache access the most and thus reduce RT the most. As all possible configurations with 2 replicas have the same write costs, WT will be the same, no matter where we add a replica.

For all further $k = l$, $2 < l \leqslant |o.RP|$, the same argumentation holds. By adding p_{jl} we add from the remaining partitions the one that has the most read operations, and thus, benefits the most from local access. As all possible configurations with k replicas have the same write costs we know that this is the best configuration with k replicas.

Finally, for all $k = l$, $|o.RP| < l \leqslant |o.WP|$, we simply add a replica to a random partition in $o.WP$ that does not yet have a replica. Read costs do not change anymore. Replication here will only be beneficial if the fact of having more writes with a local replica outweigh the costs of having an extra replica.

4.5 Managing Limited Cache Size

In this section, we consider systems with limited cache size where replicated objects have to compete with non-replicated objects for space. To illustrate the problem, assume two partitions p_1 and p_2 with a 10-object capacity each. Assume that the original distribution approach equally partitions a total of 20 objects across the two partitions. Assume further that an object o_1 assigned to p_1 is also accessed by a remote read operation r_2, that is assigned to p_2. Since the replication solution in the previous section is size-unaware, it suggests to replicate o_1 to p_2. However, there is no space at p_2 to host all of the 10 originally assigned objects and o_1. The question is which object should p_2 evict.

The best solution might be to evict one of the objects originally assigned to p_2. But it might also be that the penalty for this is actually higher than the benefit for replicating o_1, and then o_1 should be evicted, or better termed, not be replicated in the first place.

Traditional caches use standard eviction mechanism, such as evicting the least-frequently-used object as this generates the least penalty. However, in our case, calculating the penalty of an eviction is not as clear. An object that is replicated somewhere else generates less penalty than an object that is not replicated because it can be retrieved from a remote cache, which is cheaper than retrieving it from the database. Even if the object is not replicated, it might have remote reads from other partitions, which have to be considered. We have to take all this into account in our solution.

At a high level, we first assign replicas as described in the previous section without space consideration. Then we ensure in a follow-up phase that all objects assigned to a cache actually fit in the cache. For that, we evict from each cache any oversupply of objects. To do so, we calculate the expected eviction penalty for each object and evict the ones with lowest penalty. In the following, we first look at this follow-up phase for a system with no write operations, and then add write operations.

4.5.1 No Write Operations in The System

Assume partition p_i has m objects more than its cache capacity. Out of all objects some might be local-only (i.e., not replicated at any other site) while others are replicated. If p_i evicts a replicated object o, then all local read operations on o have now a costly remote read. If p_i evicts a local-only object, its own read operations as well as the remote read operations it receives from other partitions now all go to the database, again much more costly. Thus, we always have to pay a penalty. We calculate the penalty as the difference between the total expected execution time for this object if evicted and if it remains in the cache.

For a replicated object o on p_i, if o is evicted, p_i has to read the object from a remote cache. For all other p_j the situation does not change. Thus, the eviction penalty is

$$o^{EvPty} = o.n_{rp_i}(c_{rrem} - c_{rl}) \tag{4.5}$$

Clearly, the lower the number of local read operations the lower the penalty.

If a local-only object is evicted, all access has to go to the database, i.e., the local read operations but also the operations that are issued from other caches. Recall, that c_{rdb} denotes the time to read an object from the database. With this, the penalty for evicting a local-only object o on p_i is

$$o^{EvPty} = o.n_{rp_i}(c_{rdb} - c_{rl}) + (o.n_r - o.n_{rp_i})(c_{rdb} - c_{rrem}) \tag{4.6}$$

that is, both the local cache reads and any remote cache reads have to be replaced by database accesses instead. The penalty shows that the higher the number of overall reads and the more costly the database access compared to local or remote access, the higher the penalty. Furthermore, for the standard case where $c_{db} > c_{rrem} > c_{rl}$, the number of local accesses has more impact on the penalty since the time difference between database access and local access is so high.

From here, we simply sort objects by their eviction penalty and remove the m objects with the lowest penalty.

4.5.2 Write Operations in The System.

Considering writes, the removal of an object also affects the write costs.

If we remove a replicated object from p_i, then there is one less replica in the system but the write cost for a write on p_i is now c_{wrem} instead of c_{wl}. The write costs for a configuration are given in Eq. 4.3. The difference between these costs for the configuration before and after the eviction has to be simply added to Eq. 4.5. For strongly consistent protocols there is likely a benefit for writes if there is one less replica; for weak consistent algorithms it's less clear. In any case, as this is a complicated formula that has to be calculated for every configuration, sorting replicated objects by penalties can become quite costly. For the special case of random distribution of writes, however, things are simpler. For each number k of replicas, we can pre-compute the average write cost, denoted $c_w(k)$, as follows

$c_w(k) = \frac{k}{n_p} c_{wl}(k) + (1 - \frac{k}{n_p}) c_{wrem}(k))$ (right part of Eq. 4.4).

The formula only depends on system parameters such as c_{wl} and on k. With this, the penalty can be quickly calculated as

$$o^{EvPty} = o.n_{rp_i}(c_{rrem} - c_{rl}) + (o.n_w(c_w(k) - c_w(k-1)) \tag{4.7}$$

For a local-only object, in case of eviction, a write operation now only needs to update the DB with a cost denoted c_{wdb}, and no cached copies anymore. This will always be smaller than the write costs for any k of replicas. The penalty is

$$\begin{aligned} o^{EvPty} =& o.n_{rp_i}(c_{rdb} - c_{rl}) + (o.n_r - o.n_{rp_i})(c_{rdb} - c_{rrem}) \\ &+ o.n_w(c_{wdb} - c_w(1)) \end{aligned} \tag{4.8}$$

4.6 Evaluation

In this section, we evaluate our solution and compare it with other approaches using the YCSB and RUBiS benchmarks with the modifications we had discussed in Section 3.3.1. We use a total of 8 machines each having an Intel(R) 2.90GHz Dual-core, 8GB of RAM, and running with Linux OS as follows: one for the client emulator, one for the load-balancer, four application servers with caches, one database server and one for the Analyser.

Algorithms For our experiments, we compare the following approaches.

- **Obj-GP:** This is the object-based graph partitioning approach presented in the previous chapter (Section 3.2.2). Although Obj-GP does not achieve the best performance across all partitioning algorithms that were discussed in Chapter 3, we chose it as baseline because SCHISM [26] is also an object distribution algorithm that uses a similar object-graph approach.
- **RepRRO** This is the solution from Section 4.3 that uses first Obj-GP, and then replicates RRO objects.
- **RepDYN**: This is our update- and space-aware replication solution from Sections 4.4 and 4.5. It also uses Obj-GP as starting point.
- **SCHISM**: This approach is a customized version of SCHISM [26]. As discussed in Section 3.2.2, SCHISM motivated our object-based graph partitioning algorithm and was developed to partition data accessed by database transactions aiming in avoiding distributed transactions. It basically uses the object-partition algorithm that we described in Section 3.2.2 and to which we added the request distribution mechanism. SCHISM also supports replication. In contrast to our solutions RepRRO and RepDyn, replication is very tightly integrated with partitioning. More precisely, SCHISM represents each object with multiple nodes in the graph. In particular, one vertix for each request that accesses the object. Whenever the partitioning algorithm puts these nodes into different partitions, the corresponding objects will be replicated. SCHISM does not offer request partitioning. Thus, we

have added a request partitioning similar to the one used in Obj-GP. SCHISM is not cache-size aware.

4.6.1 Experiments

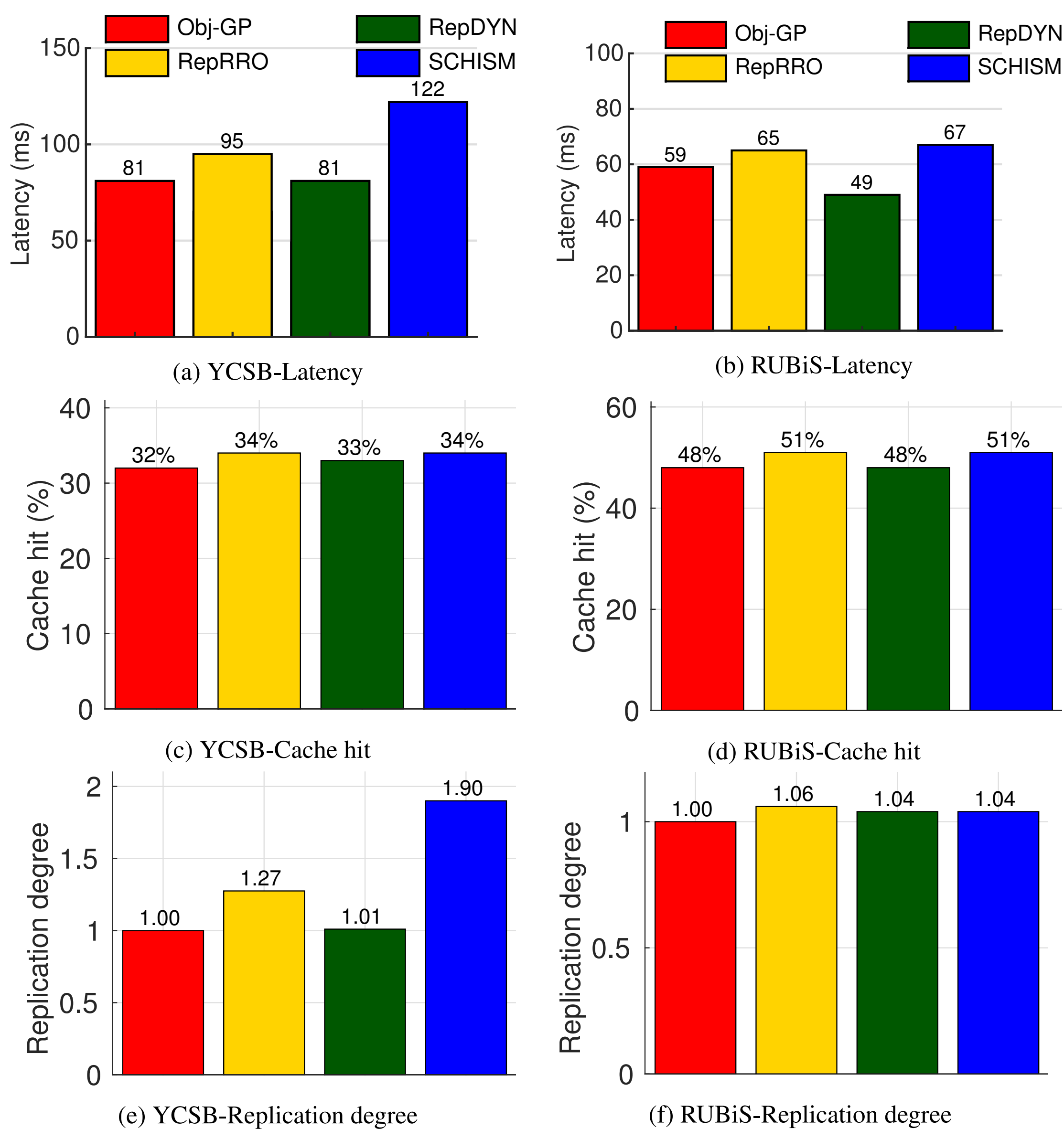
We have run experiments with various workloads, a limited cache space, and with continuous workload changes. We also show runtime analysis results.

Write dominant workload performance

In a first expirement, we adjust both YCSB and RUBiS workloads to contain 75% write requests and assume unlimited cache space. For YCSB, each scan request fetches 100 objects and each update request updates 10 objects. For RUBiS, read requests can browse several pages each with different number of items while an update request can either put a bid on an item or buy an item. In each setting we collect parameter values for 5 minutes and then run partitioning and replication. Figure 4.4 shows average request latency, local cache hit and replication degree. The latter, adapted from [125], represents the average number of physical copies for all objects. As we consider limited cache, it ranges from 0 to n (n is the number of caches).

Given the large write ratio, replication is not beneficial, therefore RepDYN replicates very few objects and performs nearly the same as Obj-GP. Both SCHISM and RepRRO replicate considerably more objects, and as a result, they have higher latency although they have more cache hits. SCHISM performs particularly bad in this scenario. With RUBiS, RepDYN is clearly the best in terms of latency, and RepRRO and SCHISM are the worst. RepDYN has the same replication degree as SCHISM, but it better targets the objects where replication does not cause too much write overhead.

The replication degrees in RepRRO and SCHISM are considerably higher in YCSB than in RUBiS. With YCSB, most requests overlap significantly with other requests, and thus, a distribution only solution leads to a lot of remote read requests, which both RepRRO and SCHISM try to avoid by replication, without properly taking update overhead into account. In contrast, RUBiS lends itself to easy distribution. Thus, replication



(a) YCSB-Latency

(b) RUBiS-Latency

(c) YCSB-Cache hit

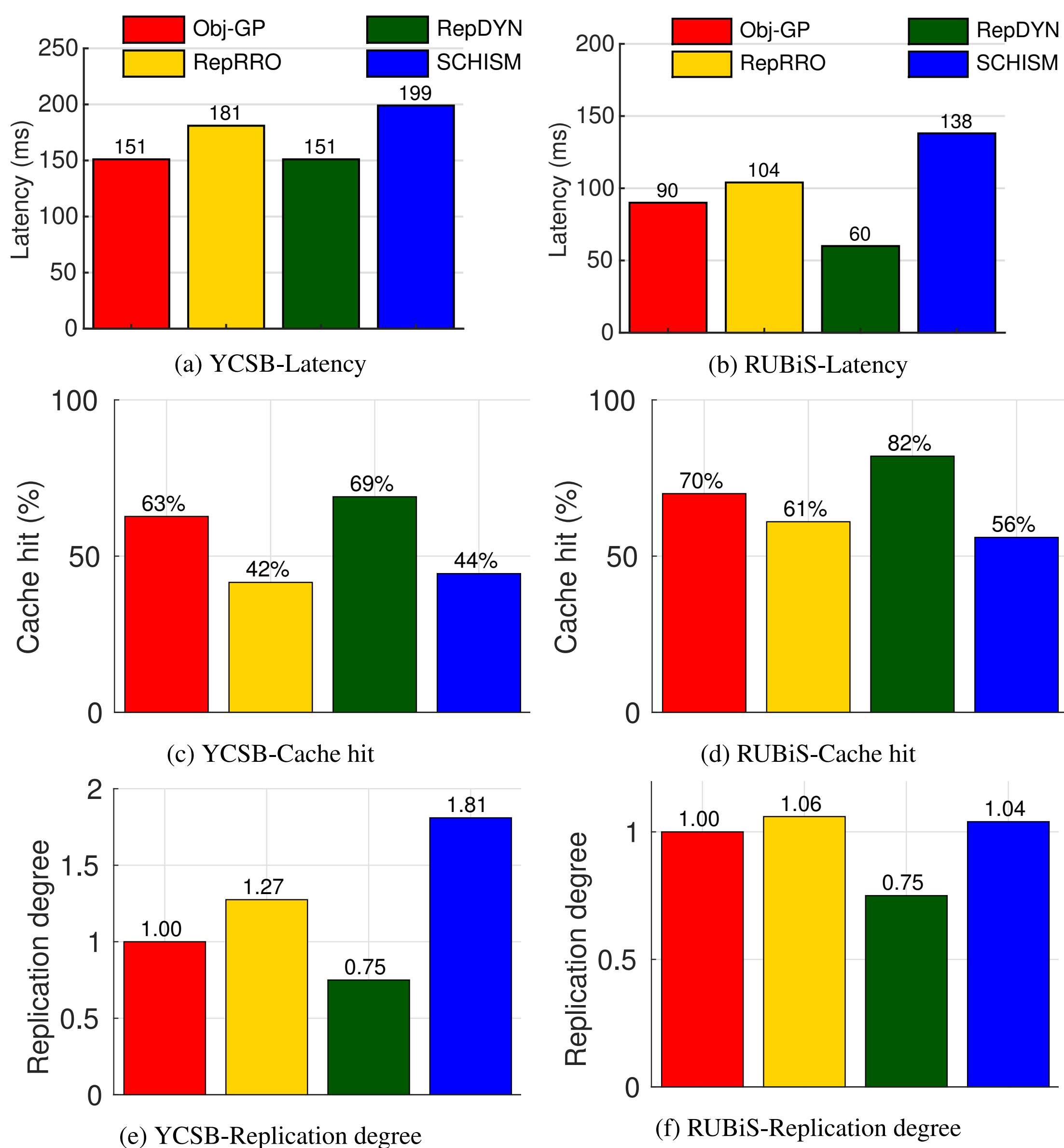
(d) RUBiS-Cache hit

(e) YCSB-Replication degree

(f) RUBiS-Replication degree

Figure 4.4: 75% write workload

is not as crucial for reads.



(a) YCSB-Latency

(b) RUBiS-Latency

(c) YCSB-Cache hit

(d) RUBiS-Cache hit

(e) YCSB-Replication degree

(f) RUBiS-Replication degree

Figure 4.5: 75% cache space

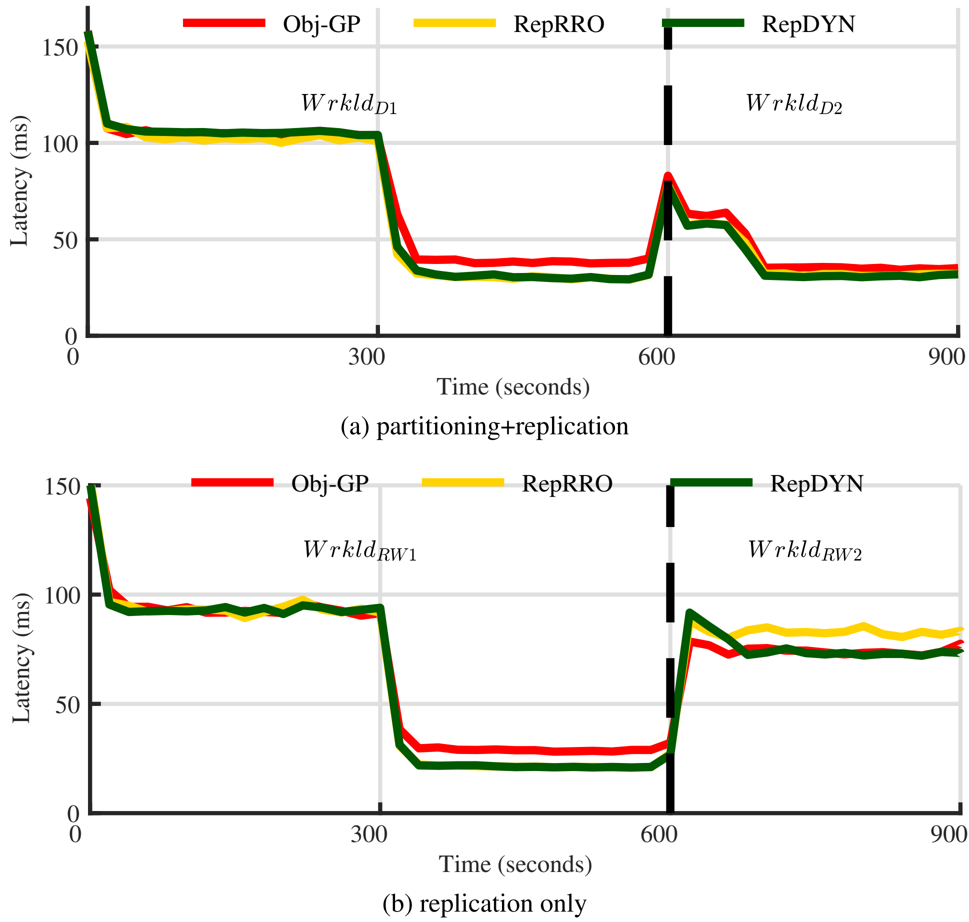
4.6.2 Limited Cache Space Analysis

In this set of experiments, we run both YCSB and RUBiS with a read-only workload. As before, each YCSB read request scans 100 items while a RUBiS read requests scans

different number of objects depending on the page it is accessing. Figure 4.5 shows the performance for YCSB and RUBiS variants with 100% read load where only 75% of the working set fits in the aggregated cache space. When looking at response times, Figure 4.5a and 4.5b, we can clearly see that RepDYN outperforms RepRRO and SCHISM. The reason is the much better cache hit ratio as seen in Figures 4.5c and 4.5d. This is due to the RepDYN's cache aware replication decision. In fact, if we look at the replication degree, Figures 4.5e and 4.5f, as RepDYN is cache aware and only 75% of the objects fit in the cache, the replication degree is 0.75. While many objects have now 0 physical copies (they were evicted), the most frequently accessed objects still have 2 or more replicas. Obj-GP, RepRRO and SCHISM have always replication degrees of 1 or higher as they are not aware of the cache size and assign more objects to each cache than the cache can actually hold. It's the cache internal policy that decides which objects to keep and which to evict. RepRRO has the same replication degree as in the previous experiment as it is neither aware of the write overhead nor the limited cache. By evicting the least important objects right from the beginning, RepDYN can achieve better cache hit ratio than the other approaches, in particular for the more sophisticated RUBiS, and thus, achieves the best latency results. RepRRO and SCHISM have particularly bad cache hit ratios, and thus, worse latency in all cases.

4.6.3 Workload Changes Triggering Partitioning

In this experiment, we change over time the workload submitted to the system. The changes are significant enough to trigger a repartitioning. Figure 4.6a shows latency over time for the three dynamic solutions: Obj-GP, RepRRO, and RepDYN. We only show results for YCSB as results for RUBiS were conceptually similar. We first run workload $Wrkld_{D1}$ of 50/50 read/write ratio and no partitioning/replication reconfiguration takes place the first 300 seconds. Then we trigger the first partitioning/replication manually after 300 seconds and turn to dynamic reconfiguration mode (whereby reconfiguration will occur if cache hit ratio changes by 10%). After further 300 seconds we switch to $Wrkld_{D2}$ that also has 50/50 read/write ratio but targets a totally different data set. Performance of all approaches is the same for the first 300 seconds (with a short



(a) partitioning+replication

(b) replication only

Figure 4.6: Dynamic reconfiguration

warm-up phase at the beginning) as no reconfiguration has taken place. After the first reconfiguration, RepRRO and RepDYN perform better than Obj-GP because the workload is read-intensive, thus both replicate many read-only objects with considerable benefit. When the second workload kicks in, reconfiguration takes place. This time again, both RepDYN and RepRRO replicate similar number of objects and thus perform better than Obj-GP.

4.6.4 Read/Write Changes Triggering Replication

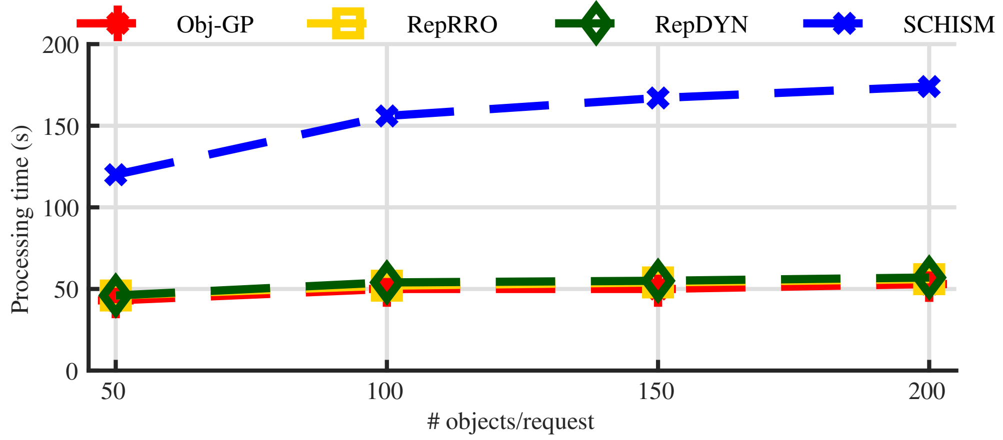
In this experiment, shown in Figure 4.6b, we run again two YCSB workloads in the same time intervals as in the last experiment. We also again switch partitioning/replication on after 300 seconds. But the changes in workload only affect the read/write ratio: $Wrkld_{RW1}$ has a 90/10, and $Wrkld_{RW2}$ a 25/75 read/write ratio. Therefore, they do not require a full repartitioning, and no reconfiguration is triggered for Obj-GP and RepRRO. In contrast, RepDYN detects the changes at 600 seconds and finds new replica assignments.

After the reconfiguration at 300 seconds, latencies are better for all approaches. Given the high read rate of the first workload, RepRRO and RepDYN have similar performance outperforming Obj-GP until the next workload kicks in because they replicate many objects. As the second workload has many writes, latencies go generally up. RepDyn removes the replicas it has previously generated and thus, performs now better than RepRRO. Its performance is now similar to Obj-GP. This shows the advantage of having a replication solution that is decoupled from the partitioning solution. If there is only a change in read/write ratios, it's not necessary to fully repartition, which is expensive. Only a light-weight adjustment of the location of replicas is needed.

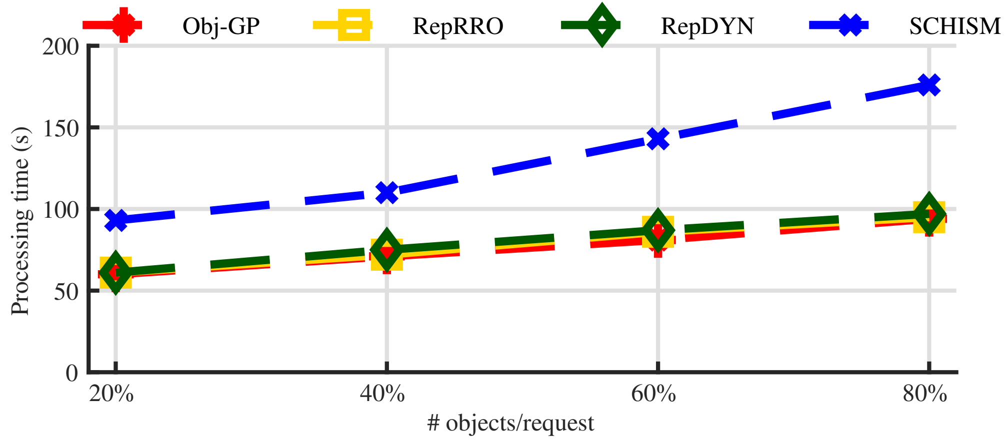
4.6.5 Solution Overhead

Finally, we look at the execution times for the different repartitioning/replication algorithms. For that, the object-graph has to be generated taking the objects and requests and their frequencies as input. The graph has then to be fed into the graph-partitioning library. For RepRRO and RepDYN, replica assignment has to be determined on top of the partitions. As the structure of the graph influences its generation, we run two workloads. Both use RUBiS with a 90/10 read/write ratio where each write request updates a single item and read requests either search items or view items and users.

In the first setup, we vary the number of objects per request from 50 to 200. Fig. 4.7a shows that for all approaches, the processing time increases with increasing number of



(a) number objects per request



(b) Overlap

Figure 4.7: Processing times

objects per request. This is directly related to the object-graph that tends to generate more edges with more number of accessed objects per request. However, the processing times for Obj-GP, RepRRO and RepDYN are similar, showing that neither of the two replication assignments has a high overhead compared to graph construction and partitioning. The processing time is steady and considerably lower than for SCHISM with the latter also increasing with increasing number of objects. The reason is that SCHISM has more vertices and edges, because if there are k requests accessing an object, SCHISM creates k vertices for this object with many additional edges between the objects accessed by the

same requests. In SCHISM, the multiple vertices are introduced specifically for enabling replication. Obj-GP, in contrast, only creates one vertex per object because it does not handle replication. RepRRO and RepDYN do it instead in a later step.

In our second experiment, each request accesses 100 objects, and there is pair-wise overlap between requests. That is, $request_1$ and $request_2$ overlap, $request_2$ and $request_3$ overlap etc. In Figure 4.7b we show processing times when the percentage of this overlap increases. Obj-GP, RepRRO and RepDYN have again similar times as the overhead for replication assignment is very low compared to the graph partitioning overhead. This time, execution times increase with more overlap as the number of edges increases significantly with higher overlap leading to more complex graphs. SCHISM's execution again takes much longer because it has many more vertices and edges.

We would again like to note that for RepRRO and RepDYN, 96% of the processing time is spent on graph-creation and partitioning, and only 4% is spent on assigning the replicas. Thus, should we choose any of the graph algorithms we discussed in Chapter 3 that has lower processing time than the Obj-GP, the overall time would be much reduced.

5

Related Work

In this chapter, we discuss several works that are related to this thesis and show how they compare to our contributions. We start by discussing related works on data partitioning and replication in the area of database systems. Then, we present works on distributed caching architectures and their respective data distribution and/or replication. Finally, we discuss some works in the area of dynamic reconfiguration for data partitioning and replication.

5.1 Data Partitioning and Migration in Distributed Database Systems

A shared-nothing architecture is a popular distribution approach that horizontally scales the database tier in order to handle increasing amount of data or load. The distributed database system comprises several stand-alone servers, or partitions, each holding a subset of data and serving transactions. A common approach is to have a front-end server as a middleware that accepts all database requests and then distributes them across the servers which is conceptually similar to the load-balancer introduced in Chapter 2 that distributes load across application servers. In the next two subsections, we show first various database partitioning techniques and then discuss some related work on database migration.

5.1.1 Data Partitioning

Data partitioning mechanisms have been widely studied in distributed systems and database communities with the goal of dividing data across a set of shared-nothing servers in order to improve scalability and performance. A good introduction can be found in [59, 83]. Based on whether the workload characteristic is taken into account when partitioning the data, partitioning mechanisms can be divided into two main categories: workload-unaware partitioning and workload-aware partitioning.

Workload unaware partitioning: These mechanisms partition the data without considering the workload access pattern or the workload transaction semantic. The most basic partitioning mechanisms are *range* and *hash-partitioning* [59]. In range partitioning, database tables are partitioned based on a value range of one of the table columns. For example, a customer orders table can be range-partitioned across servers based on the order date attribute. Hash partitioning on the other hand, assigns each row in a table, to a specific partition by hashing the value of one of its attributes, most commonly the primary key, using a hash function.

In both cases, the meta-data information that is required to know for each request which partition holds its respective data is relatively small. In range partitioning, the load-balancer requires to know the range boundaries for each partition, while hash partitioning requires a hash function in addition to information about which partition should hold which hash values. Due to their small footprint as well as their deployment simplicity, many database management systems offer one or both of these partitioning mechanisms. To name a few, MySQL, Oracel and MongoDB offer range partitioning while Dynamo [30], Cassandara [62] and Infinispan [73] use a special version of hash partitioning known as consistent hashing that requires minimal data movement whenever a partition addition/removal occurs [54].

However, despite their simplicity and minimal meta-data, both approaches have visible drawbacks such as load-impalance across partitions. Furthermore, they cause challenges for a key component of database systems which is transaction support. If the transaction is complex and updates multiple tuples residing in different partitions, the front-end server requires to run a costly consensus protocol, such as two-phase commit, to finalize such a distributed transaction resulting in a humble performance [26]. Therefore, in the last decade, many more sophisticated forms of partitioning have been proposed in the database community in order to counter the disadvantages of range and hash-partitioning.

Workload-aware partitioning: The goal of workload-aware partitioning techniques is to reduce the number of distributed transactions through grouping data items that are accessed by groups of transactions in a single partition. Thus, as data partitioning depends on particular workload transactions, these techniques are workload-aware.

In the previous chapters, we have already discussed Schism [26] and Sword [91, 61] in details. They are graph-based solutions and our approaches are inspired by them. Another work that also leverages graph representation can be found in [44]. However, rather than building a simple graph, it builds a bipartite graph that represents transactions as vertices on the left side and objects as vertices on the right. An edge is added between a transaction and its accessed objects. Then, a graph partitioning algorithm is used to

partition both transactions and objects across partitions.

As mentioned before, even though our data partitioning solutions are also based on building a workload graph, they differ in several aspects. First, we explore both request- and object-based graph partitioning. Also, we do not only distribute objects across caches, but we also distribute requests across application servers. Furthermore, Schism does not handle workload change. Sword does it in an incremental manner through monitoring the percentage of distributed transactions. Whenever this percentage exceeds a certain threshold, it moves data items across partitions to mitigate the impact of workload changes. However, they are not able to capture fine grain changes such as weight change. In contrast, we build a complete new graph on every iteration, which is feasible as we do not partition the entire database but only the most recently accessed objects and requests. This allows us to capture any form or workload changes.

Horticulture [87] uses range partitioning to partition the entire tables. It proposes several partitioning designs, each is based on selecting a certain table attribute as a candidate partitioning key. A cost model that calculates the execution costs of the various proposed designs based on a given workload is used to pick the best partitioning design. In addition to distributed transactions, the cost model also considers the workload skew factor. A workload skew happens due to a sudden load spike on a certain tuple or set of tuples. As a result, the hosted partition becomes overloaded which negatively affects the overall performance. However, instead of considering an overall skew factor, Horticulture slices the entire workload into *weighted* intervals and then calculates the overall skew factor as the arithmetic mean. As it only considers static workloads, Horticulture does not provide a mechanism for dealing with dynamic workloads that feature new queries and/or newly accessed objects. Also, the generated design may not work well with different skew access patterns.

E-store [113] targets dynamic workload skew that happens due to a sudden load spike of a certain tuple or set of tuples. It dynamically detects overloaded partitions and transfers hot tuples from their original overloaded partitions to less-loaded partitions in order to balance the load. E-store builds its solution upon the assumption that the database follows a tree-schema structure based on primary key/foreign key relationships,

and a transaction only accesses tuples within a single tree. Thus, as long as all tuples within a tree reside in the same partition, there are no distributed transactions. If a hot root tuple needs to be migrated, the entire tree has to be migrated, including also cold tuples. However, for databases that do not adhere to such a tree-schema structure and where transactions can access any tuple, E-store's skew solution will not work and lead to distributed transactions which adds to the latency.

In order to properly manage the hot tuple issue for transactions with random access patterns without introducing distributed transaction, the authors of E-store proposed a follow up approach called Clay [101]. It builds an object graph, similar to Schism, whenever an overload situation appears. This graph, however, contains only vertices and edges that are monitored during the overload situation. The reason of having this graph is to add the hot tuple's adjacent vertices to the migrated vertices set. Thus, instead of moving only a hot tuple, Clay moves also the tuples that are accessed together with the hot tuples in recent transactions to reduce the number of distributed transactions. In addition, instead of moving these vertices to a random under-loaded partition, they are moved to a partition that is not only under-loaded but also contains tuples that are likely to be co-accessed with this set of moved tuples.

Another workload-aware data partitioning [103] technique is used to transform JSON collection of documents to relational database tables. It decides, based on the observed workload, which JSON attributes (keys) should be grouped together on a partition. To do so, the technique constructs a workload graph with vertices representing attributes and edges representing attribute co-accesses. However, instead of using graph partitioning tool, they use a heuristic approach that starts from the most frequent query and assigns all its attributes to a partition.

5.1.2 Data Migration

Data migration has been playing an important role for several purposes such as recovering faulty database nodes [57, 70] or reconfiguring data across database nodes to meet workload requirements [35, 43, 104, 123]. In addition, the emerge and fast adoption of

cloud computing and virtual machines have also been leveraging data migration. For instance, in order to react to workload variations, the virtual machine in which the database resides can be migrated or the database itself can be migrated from one virtual machine to another [36, 60, 11, 28] . In addition, most works on data partitioning have utilized data migration techniques in order to move data across partitions [113, 101].

Early work about data migration can be found in the context of data replication [57, 70]. These works aim at transferring data from a non-faulty node to a recovered node that is joining back the cluster. To do so, the authors propose either to transfer the latest state copy of data items along with the ongoing transactions or transfer only the updates the recovered node has missed.

Squall [35], Morphus [43] and Parqua [104] propose data migration for database reconfiguration. These works aim to deploy a new data reconfiguration policy through migrating data across a set of DB instances in a live manner allowing both reads and writes during reconfiguration without bringing the entire system down. These works migrate individual data tuples from one DB instance to another. To do so, Squall [35] for instance, uses two approaches to migrate data. The first one is a proactive approach that starts moving data across partitions whenever the reconfiguration is triggered, and the second is a reactive approach that is triggered by a transaction to pull the tuple from the source node to the destination node if it has not been migrated yet.

Similarly, Morphus [43] targets NoSQL data stores, specifically MongoDB, and uses a pull-based approach to migrate data across nodes but in a proactive manner. A follow-up work that also uses a proactive pull-based approach is Parqua [104]. It handles reconfigurations and data movement in NoSQL data stores, such as Casandra, that use consistent hashing for data distribution.

Our data migration approach is reactive and works on a pull-based mechanism. We do not have an explicit, proactive procedure of migrating all objects whenever a new object policy is installed since such migration may negatively impact the performance of application servers. Studying the feasibility of integrating a proactive data migration approach within AdaptCache is an interesting topic of future research.

Another study [123] tries to reduce the downtime as well as the degradation phase of the migration process through completely replicating data to the destination node before discarding the source node.

In the context of cloud computing, data migration plays a vital role to enable application resource auto scaling. Auto scaling has become a major feature of cloud infrastructure capabilities. It monitors application performance and adjusts required resources to maintain steady application performance with a reasonable price. Many cloud computing providers such as Amazon AWS[1], Microsoft Azure[2] and Google Cloud[3] offer such capability. Therefore, it is important to have tools for migrating system components, including databases, across virtual machines. Examples of works that offer such tools are Zephyer [36], Rocksteady [60], ShuttleDB [11] and Albatross [28].

Zephyer [36] uses a dual mode to transfer data from a source to a destination VM. Initially, and once the migration plan is triggered, the source node pushes the meta-data to the source node in order to allow it to serve transactions. Once a transaction that is executed at the destination node requests a certain tuple that is not yet migrated, it uses a pull mechanism similar to Squall to pull the entire data page that contains this tuple from the source node. Finally, all remaining pages that have not been accessed so far are asynchronously pushed to the destination node. In principle, we follow a similar policy but we do not have the final push as we only consider "hot data" that is accessed frequently.

The same pull-based mechanism is also used by Rocksteady [60]. It forwards all client requests to the destination node and whenever the destination node receives a read query it initiates a pull request to the source node, pulls the tuples, stores it locally and then returns it to the client.

ShuttleDB [11], on the other hand, migrates the entire database to a new virtual machine by taking snapshots of the source node and applying it to the destination node. As the destination node may not be up-to-date, ShuttleDB replays queries that took place

[1]https://cloud.google.com/compute/docs/autoscaler/
[2]https://azure.microsoft.com/en-us/features/autoscale/
[3]https://cloud.google.com/compute/docs/autoscaler/

during the migration process on the destination node. This is similar to what was proposed in [57, 70].

Albatross [28] assumes a different architecture that is built on the network attached storage (NAS) paradigm whereby several database instances share the same storage but each has its own cache and serves a subset of user transactions. Thus, the problem here is not to migrate the entire database to a different node but to migrate the database instance cache in order to allow for a warm startup for a new instance. The authors propose to copy the snapshot cache of the source node to the destination node but as the snapshot might change during the copying phase at the source, the authors suggest an iterative copying where in each iteration the destination node syncs up with the source node. Once the transfer is done, transactions are then forwarded to the new instance whose cache should be in up-to-date state.

Another use case of data migration, as stated earlier, is to move data across database instances to fulfill dynamic workload partitioning. Sword [91] migrates data across nodes through an external tool that explicitly removes migrated tuples from a source node and places it at a destination node. In order not to impact performance, the data migration phase is only carried out during low demand times. E-store [113] as well as Clay [101] use Squall [35] to migrate data across nodes.

One of the challenges of the above migration mechanisms is how to deal with concurrent transaction processing that might change the state of individual tuples. In our cache, the issue is of less concern as transactions are managed through the central back-end database. A write request exclusively locks the cached object being updated and only updates it once the respective database copy has been successfully updated. If another request tries to access a locked object, the cache will forward it to the database for processing. Therefore, if the object that needs to be migrated is locked (being updated), the request on the source node has to access the database to fetch the object instead of the remote cache holding that object. Thus, AdaptCache's data migration process safely migrates objects in case of concurrent writes.

5.2 Data Replication

Data replication is a widely used approach across many storage systems. It provides availability, scalability and responsivity [89]. The idea is that a logical data item has many physical copies or replicas. In principle, a request accessing a data item can be served from any storage server that has a copy of this item. This provides availability as long as one copy is available. It provides scalability as requests can be distributed across many copies and more copies can be added as the workload increases. And replication can also provide higher system responsivity to globally distributed clients by bringing replicas close to clients [58].

Despite its benefit, data replication comes with challenges that need to be handled properly, such as keeping data copies consistent and to properly handle replica failure. These issues become even more complicated when considering replication for database systems where a transaction can access several data items and the execution across all these data items needs to be atomic, isolated and durable.

In all cases, and independently of whether data replication is implemented in a simple storage system or in a more complex database system, it can be roughly divided into two main categories: full or partial replication.

5.2.1 Full Data Replication

Full data replication means that the entire data is replicated across several nodes each of which holds copies of all data items [86, 39, 98]. In this situation, a node or a storage server is also referred to as replica. Requests are usually distributed by a load-balancer across replicas. Full data replication works well with a read-dominant workload while it struggles with write-heavy workloads because writes have to be executed on all replicas, leading to considerable overhead.

Load-balancing strategies

One simple approach to distribute requests is *round robin*. As request service times vary, a weighted version of round robin referred to as *least connections* was proposed for distributing workload requests across replicated web servers using the current number of active requests at each replica as a load measure [17]. An incoming request is dispatched to the replica with the least number of active requests.

One major advantage of utilizing such simple distribution mechanisms is their deployment simplicity as they do not require information regarding request-to-replica mapping. Also, for most workloads, they are sufficient to balance the load across replicas. However, one drawback of utilizing such simple approaches is memory contention [39]. In order to process requests, replicas usually require to perform the costly I/O operation to fetch data from disk to memory. If request instances of the same request type are processed by different replicas, then all of them must fetch the data to their memory space, leading to duplicate I/O. Furthermore, similar to the issue for stand-alone distributed caches discussed in Section 2.1, memory contention arises that leads to poor utilization of the aggregated memory space across all replicas. The operating system reacts to such a situation by performing costly memory swapping as well as additional disk I/O operations which substantially increases request latency.

Therefore, reducing memory contention across replicas by carefully sending requests to the replicas whose memory already holds their respective data is of a great performance benefit. Hence, and conceptually similar to database partitioning and the work presented in this thesis, workload-aware request distribution across replicas that aims at achieving memory access locality and equal load distribution has received considerable attention [85, 126, 97, 39, 89].

Locality aware request distribution or LARD [85] is an early approach that sends a request for an object to the server that was the most recent to serve this request. [126] follows a similar approach. A fundamental difference to our approach is that LARD assumes that each server has its own independent cache while we work with a cooperative cache that can provide a higher cache capacity. In addition, while LARD assumes that a

request accesses only a single object, our approach takes into account requests that are accessing multiple objects.

In addition, several works on database replication try to achieve transaction locality by analyzing the workload in advance and deciding about load distribution. In particular, [97] analyzes the benefits of sending a complex query to a replica where a similar query was previously sent by estimating the size of the data set overlap between these two queries depending on the predicates contained in the queries.

In the same spirit, TAashkent+ [39] statically analyzes workloads and groups transaction requests that access the same data. The requests of one group are then always sent to a specific database replica. The goal again is to enable these requests to find their working data sets in the database cache instead of performing costly disk I/O. However, as they statically analyze workload in advance, workload changes are difficult to handle. Also, as TAshkent+ targets database systems, there is the additional challenge of transaction properties.

Another load-balancing approach widely used in database systems is to separate read-only and write transactions. Ganymed [89] distinguishes between these transactions by sending write transactions to a primary replica while sending read-only transactions to other replicas. Writes are then asynchronously propagated to other replicas and reads are either forwarded to an updated replica or delayed until one of the read replicas gets the latest updates. Recent works, such as [65], follow a similar approach. The advantage of such a primary replica approach is that maintaining consistency is a lot easier than in approaches where updates can be submitted everywhere. Furthermore, the read-only replicas can be optimized for the more complex analytical queries.

Fault tolerance and availability

Data replication is also widely used for fault tolerance and to guarantee service availability. Most modern DB engines such as MySQL[4], Microsoft SQL Server[5], PostgreSQL[6] Oracle[7], MongoDB[8] as well as the Hadoop file system HDFS[9] [105] offer data replication.

The idea is to enable a data system to keep working without service interruption or with minimal performance degradation in case of a replica failure. One of the challenges for a fault tolerance system is how alive replicas can decide about the final values of data items in case of a replica failure.

5.2.2 Partial Data Replication

We have previously discussed that the complexity of current enterprise workloads makes the task of perfectly partitioning data across partitions an impossible task. Thus, replicating a few data items to partitions that need them increases access locality and also enables better load-balancing [26, 87, 61].

However, as replication increases the number of data copies, a write request to a replicated data is required to be performed at all servers containing that copy in order to guarantee data consistency [56, 58], which, as a result, adds to the transaction latency. Thus, analyzing the benefits of data replication in terms of reducing the number of distributed transactions vs. the loss such replication may induce due to the extra cost of propagating write requests has been explored within the database community.

Early work in this area [102] proposes to partially replicate data items to servers geographically close to the clients likely to access these items. This implicitly decreases the

[4]https://dev.mysql.com/doc/refman/8.0/en/replication.html
[5]https://docs.microsoft.com/en-us/sql/relational-databases/replication/
[6]https://www.postgresql.org/docs/9.1/high-availability.html
[7]https://docs.oracle.com/cd/A87860_01/doc/server.817/a76959/repover.htm
[8]https://docs.mongodb.com/manual/replication/
[9]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Data+Replication

cost of updates as updates have to be propagated to only a few and close replicas. However, in order for this solution to work, a workload should follow a certain geographical access pattern where one region will likely access the same set of data. Thus, the solution will not work for workloads that do not follow such a pattern.

Schism [26] is another work that proposes partial replication. In the previous chapter we have discussed how Schism determines which object to replicate by translating each object that is accessed by several transactions into many vertices in the workload graph, and whenever the partitioning tool puts these vertices into different partitions, the object will be replicated. Sword [91] follows a similar approach. The main issue of these approaches is that they do not take the various costs of consistency protocols into account. Moreover, as the number of transactions accessing a certain object increases, the number of vertices and edges in the workload graph becomes big. Thus, constructing and partitioning such a graph becomes expensive.

Horticulture [87] powers its partitioning mechanism by replicating read-only tables across partitions only if the partition storage has enough capacity. In order to prioritize read-only tables for replication, authors use the table temperature measurement [42], which represents the access frequency of the table divided by its size. Read-only tables are then sorted in a descendant order based on their temperatures for replication.

Rabl and Jacobsen [92] aim to achieve the best performance possible in a shared nothing database using a theoretical model that decides about either distributing or replicating data to maximize throughput and achieve equal load distribution. They use throughput as their primary objective metric and disk space as a secondary one. To replicate tables, the approach groups query types that access the same set of tables together, and based on the query weight, which is calculated as the total execution time multiplied by the respective table sizes the query accesses, it assigns queries and their respective tables to partitions in a bin-packing fashion. However, reducing the impact of data consistency in case of update queries is not explicitly considered.

All of the above solutions either use ad-hoc replication (replicate read-only objects or small tables) or their replication is tightly integrated into the distribution solution. In

contrast, our solution puts replication on top of a distribution solution and can easily handle dynamicity as the overhead of finding good replica assignments is not expensive. Furthermore, we consider limited cache size, something ignored by most approaches mentioned.

5.2.3 Data Replication in the Cloud

Cloud environments add some interesting additional issues both when data is replicated within a data center as well as when replicated across geographically distributed data centers. In addition to minimizing user request latency and balancing load across databases in a geo-replicated environment [81, 68], other research is looking into the price of using cloud resources as well as energy consumption [15]. The authors propose to replicate data across servers within the same data center and/or across different data centers in order to enhance performance and, at the same time, reduce energy consumption. The simulated results show that integrating energy saving as a replication objective can have a potential impact on energy saving. However, this comes at the price of performance.

Spanner [24] is a more dynamic data storage system that allows for fine-grained data replication across data centers based on several criteria related to user's latency as well as the cost of maintaining consistency across replicas. However, unlike our replication approach, the number of replicas as well as their locations are manually determined. A conceptually similar approach can be also found in [47]. Surveys on data replication tehniques in cloud data centres can be found in [96, 76]

5.2.4 Data Consistency for Replicated Data

The benefits of replication for scalability and performance depend on the used consistency algorithm. In general, consistency protocols can be divided into two main classes depending on when copies are updated [83]. The first one is called lazy or weak, and the second one is called eager or strong. This categorization holds for both data storage systems and database systems. However, due to transactional requirements, consistency

protocols for database systems are more complicated.

Lazy replication

Lazy replication requires to execute the update on one replica and then, after committing and returning the result to the user, propagate the update to the rest of the replicas. The strength of lazy replication is its performance as no communication for update propagation occurs within the response time observed by the client.

Early works for transactional databases focused on primary-copy lazy consistency where a primary node is in charge of executing all updates and only propagate them to the secondaries after informing clients [31, 29]. This is also the approach used in Ganymed [89] that we discussed earlier. More recent systems such as PNUT [22] extend that lazy replication model to have a dynamic primary node instead of a predefined fixed primary node. However, due to the immense amount of processed data, PNUT relaxes the transactional requirements in favour of performance.

A weak, yet wildly adopted correctness model of lazy replication is eventual consistency. It requires that if no further updates are received, all copies will be eventually consistent [9]. Several cloud storage systems such as Amazon Dynamo [30] and Google Cloud Datastore use eventual consistency as their consistency level. However, as eventual consistency makes no transactional guarantees, it is not suitable for transactional systems.

Eager replication

Eager protocols ensure that all replicas execute the update before returning to the client. They are widely used in fault tolerance systems. Conventional database systems with transactional requirements usually uses Two-Phase Commit protocol to guarantee that all data copies have the same value at the end of transaction. Should a replica fail after having committed a value, all available replicas will also commit. Non-transactional storage systems use Paxos [64] or a modern version of it called Raft [82]. Paxos is a widely adopted consensus protocol that allows a set of alive replicas to form a con-

sensus over a value in the presense of replica failure. Several large scale data storage systems such as Google Megastore [10] and Spanner [24], that are used to store various Google products, utilize Paxos to build a fault-tolerance storage system. A comprehensive comparison between Two-Phase Commit and Paxos can be found in [45]. In contrast to Paxos, Raft [82] decomposes the consensus challenges into relatively independent sub-problems and solves each of them individually which makes it easier to understand and also implement than Paxos.

As eager replication can have higher latency compared to lazy replication, several works try to mitigate such performance degradation [56, 38]. In [56], the authors propose a strong consistency implementation without 2PC by propagating changes to other replicas before committing the transaction using a total order group communication protocol but committing the transaction and return to the client before the other replicas have applied the changes. In [38], the authors allow the front-end middleware rather than individual database replicas to determine the ordering of the committed transactions and send the ordering of the transaction execution in one shot to different replicas. This can result in a considerable performance benefit.

Recent works analyze the trade-off between different consistency levels with the goal of dynamically picking the best consistency level protocol [20, 67, 110]. Although the goal of maximizing performance gain is the same as ours, a major difference to our replication solution is in the way the problem is tackled. These works assume a fixed object replication policy in place and the system can switch to a different consistency protocol to achieve the best performance possible. In contrast, our solution assumes a cache with a given consistency protocol and the goal is to pick the best replication policy under the given consistency algorithm.

5.3 Data and Space Management for Distributed Caches

Caching can be implemented either as a stand-alone architecture such as Memcached[10] or it can be integrated with the application tier such as Ehcache[11]. Based on the given architecture, we will discuss related works on data distribution, data replication, and space management in distributed caching.

5.3.1 Cache Architectures and Data Partitioning

Many current key-value caching systems rely on a dedicated cluster of cache nodes to store recently used data. Example of these caching systems include Memcached and Redis[12]. As the cache nodes are completely decoupled from the application servers, all cache calls are remote. Our Cooperative Cache overcomes this limitation by coupling each application server with its local cache and, at the same time, letting all local caches share their data.

To distribute data among nodes, many caching systems, such as Memcached and Redis, utilize consistent hashing [54]. As described previously, consistent hashing is a widely used distribution technique that allows for dynamic node changes without the need to reorganise all the data.

Chord [111, 112] is a consistent hashing protocol that allows for fast key lookup and minimal data movement when the number of nodes changes. It does so by creating an identifier space as a logical ring and distributing both nodes and data items across the ring space by hashing their ids and keys respectively. Each node stores all data items with keys that are in the range between its own and its predecessor's identifier. Whenever a key is requested, the node that receives the request forwards it to a node whose id is closer to the requested key using a routing table that is built in a way that greatly reduces the number of hops. A new node is placed within the ring based on its hashed id and receives part of the data of its predecessor in the ring. Thus, only a small portion of the

[10]https://memcached.org/
[11]http://www.ehcache.org/
[12]https://redis.io/

data has to be moved to the new node and there is no need to reshuffle the entire data in case of node changes.

One issue of the Chord design is that it does not guarantee even hash values distribution over the hash space, which can lead to load and size imbalance [6, 49]. Also, when a new node is added it alleviates only the load of its predecessor in the ring. Hwang and Wood [50] adjust the hash space by representing each physical node as many virtual nodes. All data items of a physical node are virtually distributed across all its virtual nodes, and it is the virtual nodes that build the ring. When a new node is added, its virtual nodes will be neighbours of various existing virtual nodes that reside on different physical nodes, thus, the load from many different physical nodes will move to the new node. Individual virtual nodes can also be moved from one physical node to another. This allows for better load-balancing as well as minimal data shuffling whenever the number of nodes changes.

In addition, other works have targeted the memory allocation mechanism that Memcached uses to store objects. In Memcached, whenever an object is cached it will be assigned to a slab, where slabs of equal sizes are grouped into a class. If, for example, all objects are of the same size, they will be assigned to the same class which makes other classes not well utilized resulting in imbalanced utilization. Lama [48] optimizes the memory allocation mechanism by calculating the best number of slabs within each class as well their sizes based on the workload. The entire hash space is then restructured in order to maximize memory cache usage.

Our cache partitioning solution is quite different. It is able to detect load-imbalance during run time as it occurs and initiates new partitioning according to the most recent workload.

Other works that target data partitioning across a set of application server caches can be found in [118, 119]. The authors assume a front-end load-balancer that dispatches requests across a set of servers each of which contains an independent caching component to store the recently accessed objects that are fetched from the backend storage. In order to reduce backend access, the authors propose a request-policy that is used to

send a request to a server whose cache contains the requested data. Instead of building a fine-grained request policy similar to ours, they build a coarse-grained policy that maps a category of requests to a server or set of servers. Whenever a request arrives, the load-balancer assigns its respective server through its category.

Again, individual caches work independently. Hence, a request updating a certain object at a certain server may cause inconsistency with other cached copies of the same object at different caches. In addition, the proposed solution assumes that a request is targeting a single object, which, as we have previously shown in Chapter 3, is not always the case. Thus, distributing only requests without their respective objects will not achieve the best outcome.

In a quite different context, non-uniform memory access or NUMA [63, 69], has some similarity with our cooperative cache architecture. In NUMA, cores on different nodes can access remote memories as well as their local memories. As accessing local memory is faster than accessing a remote one, memory segments have to be allocated among different NUMA nodes in a way that keeps remote memory access infrequent. In [63], two memory allocation policies are discussed. The first one is *local*; memory pages are allocated to memory attached to the core that is processing its code while the second is *random* where memory pages are assigned to memory cores in a round-robin fashion. However, such simple memory allocation policies still require processes to access remote memory pages. Thus, we believe that the data partitioning and replication solutions presented in this thesis could inspire a dynamic and application-aware NUMA memory allocation solution.

Shared-everything is another architecture that might benefit from our data partitioning solution. In shared-everything, the entire database is located at a multi-socket, multi-core server. ATraPos [90] indicates that the challenge of scaling such a multi-core system is that it experiences bottlenecks in terms of accessing centralized data structures, such as shared locks and the list of active transactions. ATraPos dynamically partitions these centralized data structures as well as their receptive data across cores in a way that reduces inter-core communication and balances the load across cores. However, as data is partitioned across different cores but within the same machine, data migration is not seri-

ously considered. In addition, data replication to enhance data locality for more complex workloads is not considered.

Another scalability issue in a shared-everything architecture is the coherence of the cache directory that keeps information about locations of data across memory cores. Whenever a change is made to the content of any core's memory, the centralized directory has to be updated accordingly, which as a result, can cause scalability issue. A previously proposed solution [33] aims at moving the centralized data directory to an in-node directory where data location information is located within each node. This is in order to reduce the amount of exchanged messages when reading or writing data.

5.3.2 Data Replication and Caching

Replicating data across caches can be used to balance the load across caches and to achieve fault-tolerance. Replication for these purposes has been developed in architectures that decouple cache servers from application servers [111, 6, 4, 7]. As these caching systems usually use consistent hashing where an object is assigned to a single node, a failure of a cache instance can harm the performance. Thus, several works and systems try to bring high availability to these caching systems [111, 50].

An example of work that utilizes object replication to solve load imbalance across Memcached nodes can be found in [124]. The proposed solution detects keys with high popularity and replicates them across underloaded partitions. Requests are then dispatched to nodes using a load-balancer that maintains information about replicated objects.

Repcached[13] is an extended replication library that replicates data across Memcached instances. It provides a flexible replication mechanism that allows developers to choose the type of operations to replicate at a peer cache. Other caching systems that offer object replication include Oracle's TimesTen In-Memory Database Replication[14], IBM Web-

[13] http://repcached.lab.klab.org/

[14] https://docs.oracle.com/cd/E11882_01/timesten.112/e21635/overview.htmTTREP115

Sphere eXtreme Scale [51], and the recent Amazon DynamoDB Accelerator (ADX)[15] that works on top of DynamoDB [30] to provide faster data access.

In caching systems where cache instances are collocated with application servers, replicating data across caches aims also to enhance data access locality. This is a typical case of Ehcache[16] and JBoss cache[17].

Compared to our replication solution, all of the above replication mechanisms do not take into consideration the update cost of replicated objects. In addition, they do not consider the problem of the limited cache space. We believe, however, that their flexible replication APIs allow for easy integration of our replication solution.

5.3.3 Managing Cache Space

The problem of managing cache space across several virtual machines (VM) has received considerable attention. At the abstract level, the architecture comprises a set of applications each running within a single VM and all VMs share a common cache space. The question becomes how to dynamically divide the cache space across these applications in a way that maximizes the overall performance [75, 4, 93].

CloudCache [4] partitions cache space across a set of applications based on their demands. In addition, it tackles the issue of a limited cache space by migrating, if necessary, both the VM and the attached cache space to a different host with enough cache capacity. Multi-Cache [93] divides hierarchical cache layers, each with different speed, among a set of VMs and the goal is to achieve the best allocation policy.

A conceptually similar research problem is how a set of applications that share a common cache can divide the cache space among the applications based on their demand [109, 21, 16]. Moirai [109] divides a shared cache space across applications in a way that can either prioritize certain applications or maximize the overall application hit ratio. Memshare [21] adds more memory to applications with high demand while cutting

[15] https://aws.amazon.com/dynamodb/dax/
[16] http://www.ehcache.org/
[17] http://jbosscache.jboss.org/

off memory of over-provisioned applications. The goal is to reduce the costly cache eviction process. mPart [16] is a recent work that also tries to achieve the same goal using a theoretical model that uses as an input the cache hit ratio curve for each application as well as the entire cache capacity in order to find an optimal cache allocation strategy.

Managing cache space can be also useful for modern multi-core hardware systems where database queries can compete on the cache space and harm performance. In [80], the authors propose to distribute the cache across queries in a way that achieves the best performance and avoids cache competition across queries. They dedicate certain amounts of cache to queries based on their type according to a priory fixed analysis of queries' cache requirements.

Clearly, the goal of all of the above works is different to the problem of managing the cache space this thesis deals with. A work close to ours can be found in [66]. The authors assume a limited storage space and the problem is to decide either to replicate an object to that storage or not. While they use a formula that weights objects based mainly on their access count, they do not differentiate between various access costs for various request types. Furthermore, they do not consider the write overhead.

5.4 Dynamic System Configuration

Research on dynamic system configurations aims to produce a highly adaptive system with minimal human interaction. The need for a dynamic approach has increased with the advent of cloud computing and virtualization. As stated earlier, cloud computing allows several VMs to run on a single machine. Thus, these VMs can be automatically configured in order to distribute the physical resources of the host machine among the different applications according to their workload variations [106, 84, 94, 108]. Even within each application, there is a space to control the service level according to the workload against different resources (CPU, memory) [88, 18]. Alternatively, several machines can be dedicated to a large-scale enterprise. This can be dynamically self-configured through allocating extra machines in the peak workload time while deallocating some machines during low demand time [37, 117, 116, 19]. A more recent work [107] combines the

above two approaches. It dynamically allocates resource for applications whose services are hosted in heterogeneous cloud platforms by either changing the number of VMs, resizing the existed ones or both. To facilitate cloud resources provisioning, workload modeling for applications can be used [12, 52].

Our partitioning and replication caching solution is also dynamic. The AdaptCache tool is eager to work with any application that is hosted on top of a cooperative cache. Second, the tool is orthogonal to the targeted system in terms of number of machines, types of the network, used consistency protocol and various types of workload changes. All these system-related parameters and performance metrics are transparently extracted through the developed tool.

6

Conclusions

The Multi-tier architecture is such as the widely popular system design pattern that divides the entire system into many layers; web layer, the application layer and the back-end database layer. Each of these layers is logically and likely physically separated to allow for better scalability, availability and portability. An application tier, for example, can be scaled to have several application servers to serve increasing user demands. In such a case, a load-balancer is placed at the front of these instances to intercept user requests and distribute them across the application instances.

Data intensive applications require to fetch data from the back-end database to serve client requests. To alleviate load on the back-end data tier, a caching module can be added to each application instance. However, to fully utilize the entire cache space, a cooperative cache that connects all application local caches is constructed. The question becomes how to manage requests and objects in such architecture to achieve best performance possible.

This thesis provides novel approaches for conducting request and object partitioning as well as replication for cooperative caches. The approaches are adaptive, requiring no external interaction to function. They transparently monitor the workload to detect changes and react accordingly.

To motivate the need for such an adaptive partitioning approach, we first analyzed a

real workload to extract important request and object access patterns. Out of this analysis, we concluded important aspects that impact our caching partitioning solutions. In particular, we found that request types change their accessed objects over time. Also, we found that different objects have different popularities as well as different access times. These observations, along with others, influenced our data partitioning solutions.

We developed a suite of partitioning solutions that aim in distributing requests and objects across servers in a way that achieves high local data access and an equal load distribution. We explore two solution spaces that utilize both graph and hyper-graph data structures to represent the workload. The first solution space is *request distribution first* that builds a request graph/hyper-graph and then partitions it, using a graph/hyper-graph partitioning library, into a number of partitions each assigned to a server. The respective objects are then distributed across these partitions in a way that minimizes remote cache access. The second solution space is *object distribution first*, which builds an object graph/hyper-graph, partitions it, and then distributes respective requests.

Due to the complexity of enterprise workloads that prevents full local data access, we empowered the basic partitioning solution with a novel dynamic object replication. The replication approach adds object replicas to partitions that need them, thus making read requests to these replicas fast. It also considers the challenge of maintaining a consistent object state across replicas in case of write requests as well as the challenge of limited cache space. To do so, the replication solution evaluates the performance of an object as the gain of replicating an object and the overhead such replication may induce by maintaining a consistent state across replicas in case of write requests. Also, in case of limited cache space, the solution decides to evict objects, either replicated or not, that are least beneficial to performance.

We further extended the partitioning and replication solution to work in an adaptive manner. That is, it tracks workload changes and reacts quickly. The solution tracks two types of workload changes. The first one occurs when the local cache hit ratio changes by a predefined threshold, which triggers full object and request partitioning in order to improve cache hit ratio. As this re-partitioning step may induce object migration across caches, which harms the performance, we assign each partition to a cache in a way

that keeps as many objects as possible at their caches. The second workload change type occurs when the ratio of read/write objects changes, which only triggers object replication in order to improve user latency. We also optimize the solution processing time by ignoring requests and objects with least performance benefit.

We have implemented our partitioning and replication solution into a tool called AdaptCache. AdaptCache is capable of performing data partitioning and replication for applications that use a cooperative cache framework. AdaptCache needs no prior knowledge of the application code nor the specific configurations such as the number of servers or the consistency requirements. Rather, it observes the workload using minimal amount of data and decides to dynamically partition and/or replicate objects. We have conducted extensive experiments using RUBiS and YCSB benchmarks with various combinations of workloads. The results show that our solution can enhance performance for real workloads. In addition, we compare the performance of various partitioning solutions against several workloads and analyse their results. The result also shows that our replication solution can greatly enhance the performance in case of write requests and limited cache space. In addition, the result shows that our replication solution outperforms a well-known replication solution.

7

Future Work

In this chapter we outline future research lines arising from the work carried out in this thesis. In particular, we discuss how the cooperative cache can be compared to other caching architectures such as a stand-alone cache. In addition, we discuss the possibility to optimize the log messages of the cooperative cache. Another research direction is how cache instances can auto-scale based on the workload demands. Also, we discuss the possibility to empower the current caching system with a proper fault-tolerance mechanism. We finally discuss how caching solutions presented in this thesis can be applied to systems with a similar architecture to the cooperative cache.

7.1 Cooperative Cache vs Stand-alone Cache

As a stand-alone distributed caching architecture that caches data in dedicated cache nodes, such as Memcached, is widely used in web enterprises [124, 53], its performance compared to co-allocated distributed caching that caches data within application nodes, such as our cooperative cache, requires an in-depth analysis. The idea is to compare the performance of these two caching architectures with respect to various parameters such as the load on the system, the number of nodes, the cached data size, and different mixtures of read/write workloads. Another metric that can be analyzed is how the deployed consistency protocols impact the performance in both scenarios in case of replicated cached objects.

One can imagine that for scenarios where the individual instances of the cooperative cache have enough memory to hold their working data set as well as enough processing power, the cooperative cache can outperform a stand-alone caching system as it has more local accesses. However, if any of these two resources becomes saturated, a stand-alone caching might perform better. A comprehensive comparison between these two architectures can reveal such interesting observations.

In addition, most cloud providers offer caching solutions with a similar architecture to the standalone Memcached. Examples of these cloud caching systems include Amazon DynamoDB Accelerator (DAX)[1], Microsoft Azure Redis Cache[2] and Google Memorystore[3]. These caching systems are dedicated to the provider's services and designed to work exclusively with their hosted applications. Although they allow little freedom for users to control object locations, comparing these solutions to our cooperative cache is another viable research track. The idea again is to compare these caching architectures regarding performance in addition to the configuration and management costs.

[1]https://aws.amazon.com/dynamodb/dax/
[2]https://azure.microsoft.com/en-us/services/cache/
[3]https://cloud.google.com/memorystore

7.2 Optimizing Log Messages

Log messages are a central part of our caching solution. They are used for the two main purposes of workload monitoring and carrying out data partitioning and replication. Although having them is a non-avoidable design option, log messages can degrade performance of the targeted application, which is mainly due to the additional processing that has to be carried out by both the log sender (the application servers) and the log receiver (the Analyser). In addition, these messages consume network bandwidth. While this may not be an issue for a system that is hosted on premise with enough network bandwidth, it can cause extra charges for cloud-hosted applications.

Therefore, minimizing the logging overhead can result in a considerable performance benefit as well reduced costs for cloud-hosted systems. Options to optimize log message management include decreasing the number of log messages, decreasing the size of messages or both. Conducting such an optimization step, however, has two main challenges. First, making sure that the reductions in message size and/or number is not negatively impacting the quality of the overall workload meta-data which might decrease the accuracy of workload monitoring or, more importantly, the quality of the produced partitioning/replication solution. Second, the log message optimization step itself should not incur an extra overhead at the sender or the receiver.

7.3 Autoscaling Cache Nodes

The load experienced by the system varies greatly by time [100, 113]. If the system experiences low load, a few servers can be enough to serve the workload. But if the load is high, more nodes might be needed to avoid overhead and performance degradation. Therefore, a server auto-scaling mechanism that aims to either increase or decrease the number of active servers, depending on the currently served workload, is necessary.

There are several challenges to conduct such auto-scaling. First, a proper workload and/or resource tracking mechanism is needed that is able to detect such overload/underload situation. Second, proper reconfiguration algorithms are needed that are able to add

or remove nodes without violating consistency requirements. Third, this reconfiguration should be transparent to the user, which requires executing the action without service interruption and with minimum performance degradation. Thus, an important technique to minimize such degradation during the auto-scaling phase is to keep the amount of requests and objects that need to be migrated across servers low.

7.4 Fault Tolerance

In addition to providing low-latency for user requests, achieving high availability is another important feature that can be added to the cooperative cache architecture. Several components of AdaptCache and its extension for replication, such as the load-balancer, the cache instances and the Analyser, are vulnerable to failure. These failures can happen due to either hardware failures such as network problems or software failures such as running out of memory and excessive disk I/O operations.

We can categorize the components based on their failure impacts into two categories. The first category comprises components where a failure brings down the entire system. For AdaptCache, this is mainly the load-balancer. However, a simple restart with an initial round-robin distribution technique should be sufficient to get the system working again. The next rebalancing will again provide the load-balancer with a new policy. Thus, making it highly available through, for example, replication, is important.

The second category comprises components where a failure will not shut down the entire caching system but rather will degrade its performance. These components include the application server instances as well as the Analyser. Having a failed server instance can degrade the performance because it will not be able to serve requests, and because its cached data becomes unavailable. However, as the load-balancer can monitor application instances, it can disable the failed one and redirect requests to alive servers. But these have to load the data originally cached in the failed server from the back-end database. This phase of loading the failed data causes a temporary performance degradation. Also, as this newly loaded data are not well partitioned, performing a new partitioning may be necessary to achieve a better performance.

In case of failure of the Analyser, the system looses its capability of performing workload monitoring as well as carrying out data partitioning/replication, which can lead to reduced performance over time when workload changes occur. A high availability solution either provides replication/persistence of the Anlyzer's meta-data to allow quick and complete take-over by a new Analyser instance or a new instance would have to start collecting meta-data if starting from scratch.

In all cases, a high availability solution needs to be able to detect any component failure fast and then react to such failure in a way that minimizes the performance degradation period and, if possible, avoids the downtime for the entire system. To track cache instances aliveness, many performance metrics that can be used are already captured in our design. Alternatively, tools to detect performance changes, such as [13], can be used.

7.5 Applying Caching Solutions to NUMA

Although the caching solution presented in this thesis is mainly designed for a cooperative cache system in multi-tier architectures, other systems that share conceptually a similar architecture can benefit from our solutions. One potential example of these systems is the non-uniform memory access or NUMA system [63, 69]. NUMA allows several computational nodes, each with its own CPU and memory, to access other nodes' memories. This architecture is being adopted by many operating system vendors such as HP[4], Dell[5] and VMware[6]. Thus, many applications can enjoy this performance boosting technology by allowing processes to access remote memories.

Conceptually similar to the cooperative caching, a careful distribution of processes across nodes and data across memories will result in a high data access locality. Also, replicating popular data, in particular those that are rarely updated, across memories that require them may further enhance performance. Thus, some of the ideas presented in this thesis might be applicable to NUMA. However, NUMA has different challenges.

[4]https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/c03261871$_2$012.*pdf*
[5]http://doc.xueqiu.com/14738e1820f33fea1ba083fd.pdf
[6]https://pubs.vmware.com/vsphere-4-esx-vcenter/index.jspcom.vmware.vsphere.resourcemanagement.doc_41/

In particular, it deals with memory pages [78] rather than high level objects. This requires a mechanism to have full control over memory pages and their locations. Furthermore, NUMA deals with processes rather than requests, which requires a mechanism to reschedule these processes to different CPUs.

Bibliography

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[2] S. Ali. Contquer: an optimized distributed cooperative query caching architecture. Master's thesis, McGill, 2009.

[3] I. Arapakis, X. Bai, and B. Cambazoglu. Impact of response latency on user behavior in web search. In *SIGIR*, 2014.

[4] D. Arteaga, J. Cabrera, J. Xu, and S. Sundararaman. Cloudcache: On-demand flash cache management for cloud computing. In *USENIX*, 2016.

[5] O. Asad and B. Kemme. Adaptcache: Adaptive data partitioning and migration for distributed object caches. In *Middleware*, 2016.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, 2012.

[7] X. Bai, I. Arapakis, B. Cambazoglu, and A. Freire. Understanding and leveraging the impact of response latency on user behaviour in web search. *ACM Transactions on Information Systems*, 36(2):21:1–21:42, 2017.

[8] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[9] P. Bailis, A. Ghodsi, J. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, 2013.

[10] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[11] S. Barker, Y. Chi, H. Hacigumus, P. Shenoy, and E. Cecchet. ShuttleDB: Database-aware elasticity in the cloud. In *ICAC*, 2014.

[12] A. Bhattacharyya, S. A. J. Jandaghi, and C. Amza. Semantic-aware online workload characterization and consolidation. In *IEEE CLOUD*, 2018.

[13] A. Bhattacharyya, S. Sotiriadis, and C. Amza. Online phase detection and characterization of cloud applications. In *CloudCom*, 2017.

[14] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] D. Boru, D. Kliazovich, F. Granelli, P. Bouvry, and A. Zomaya. Energy-efficient data replication in cloud computing datacenters. *Springer Cluster Computing*, 18(1):385–402, 2015.

[16] D. Byrne, N. Onder, and Z. Wang. mPart: Miss-ratio curve guided partitioning in key-value stores. *ACM SIGPLAN Notices*, 53(5), 2018.

[17] V. Cardellini, M. Colajanni, and P. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[18] G. Casale, A. Kalbas, D. Krishnamurthy, and J. Rolia. Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. In *Middleware*, 2009.

[19] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *ICAC*, 2006.

[20] H. Chihoub, S. Ibrahim, G. Antoniu, , and M. Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *CLUSTER*, 2012.

[21] A. Cidon, D. Rushton, S.Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *USENIX ATC*, 2017.

[22] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[23] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[24] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[25] A. Crescenzi. Time pressure in information search. In *SIGIR*, 2015.

[26] C. Curino, Y. Zhang, E. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, 2010.

[27] W. Dakka, L. Gravano, and P. Ipeirotis. Answering general time-sensitive queries. In *CIKM*, 2008.

[28] S. Das, S. Nishimura, D. Agrawal, and A. ElAbbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. In *VLDB*, 2011.

[29] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB*, 2006.

[30] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, , and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOPS*, 2007.

[31] P. Minet E. Pacitti and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *VLDB*, 1999.

[32] Ehcache. Ehcache Replication Guide. `http://www.ehcache.org/`.

[33] N. Eisley, L. Peh, and Li Shang. In-network cache coherence. In *MICRO*, 2006.

[34] A. Elmore. *Elasticity Primitives for Database as a Service*. PhD thesis, University of California, Santa Barbara, 2013.

[35] A. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. ElAbbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *SIGMOD*, 2011.

[36] A. Elmore, S. Das, D. Agrawal, and A. Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, 2015.

[37] S. Elnikety, S. Dropsho, E. Cecchet, and W. Zwaenepoel. Predicting replicated database scalability from standalone database profiling. In *EuroSys*, 2009.

[38] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, 2006.

[39] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *EuroSys*, 2007.

[40] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. In *SIGCOMM*, 1998.

[41] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[42] E. Boughter G. Copeland, W. Alexander and T. Keller. Data placement in Bubba. In *SIGMOD*, 1998.

[43] M. Ghosh, W. Wang, G. Holla, and I. Gupta. Morphus: Supporting online reconfigurations in sharded NoSQL systems. In *ICAC*, 2015.

[44] L. Golab, M. Hadjieleftheriou, H. Karloff, and B. Saha. Distributed data placement to minimize communication costs via graph partitioning. In *SSDBM*, 2014.

[45] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transaction on Database Systems*, 31(1):133–160, 2004.

[46] J. Gray, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[47] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB*, 2014.

[48] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *USENIX ATC*, 2015.

[49] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. Freedman, K. Birman, and R. Renesse. Characterizing load imbalance in real-world networked caches. In *HotNets*, 2014.

[50] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *ICAC*, 2013.

[51] J. Marshall and J. Pape and K. Peterson and G. Reid and F. Santos and B. Silva and F. Senese. WebSphere eXtreme Scale V8.6 Key Concepts and Usage Scenarios. `http://www.redbooks.ibm.com/redbooks/pdfs/sg247683.pdf`.

[52] S. A. J. Jandaghi, A. Bhattacharyya, and C. Amza. Phase annotated learning for apache spark: Workload recognition and characterization. In *CloudCom*, 2018.

[53] X. Jin, X. Li, H. Zhang, R. Soule, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP*, 2017.

[54] D. Karger, E. Lehman, T. Leighton, M. Levine, D Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Symposium on Theory of Computing*, 1997.

[55] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for rregular graphs. *Parallel and Distributed Computing*, 48(1):96–129, 1998.

[56] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *VLDB*, 2000.

[57] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *International Conference on Dependable Systems and Networks*, 2001.

[58] B. Kemme, R. Jimenez-Peris, and M. Patino-Martinez. *Database Replication*. Morgan and Claypool Publishers, 2010.

[59] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[60] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *SOSP*, 2017.

[61] K. Kumar, A. Quamar, A. Deshpande, and S. Khuller. SWORD: Workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal*, 23(6):845–870, 2014.

[62] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[63] C. Lameter. NUMA (Non-Uniform Memory Access): An overview. *ACM Queue*, 11(7):1–12, 2013.

[64] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[65] J. Lee, S. Moon, K. Hwan Kim, D. Kim, S. Cha, and W. Han. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. In *VLDB*, 2017.

[66] M. Lei, S. Vrbsky, and X. Hong. An on-line replication strategy to increase availability in data grids. *Future Generation Computer Systems*, 24(1):85–98, 2008.

[67] C. Li, J. Leitao, A. Clement, N. Preguica, N. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX*, 2014.

[68] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguic, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[69] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[70] W. Liang and B. Kemme. Online recovery in cluster databases. In *EDBT*, 2008.

[71] C. Luo, X. Li, Y. Liu, T. Sakai, F. Zhang, M. Zhang, and S. Ma. Investigating users' time perception during web search. In *CHIIR*, 2017.

[72] C. Luo, F. Zhang, X. Li, Y. Liu, M. Zhang, and S. M. Manipulating time perception of web search users. In *CHIIR*, 2016.

[73] F. Marchioni and M. Surtani. Infinispan data grid platform. *PACKT Publishing*, 2012.

[74] R. Maredia. Automated application profiling and cache-aware load distribution in multi-tier architectures. Master's thesis, McGill, 2011.

[75] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vCacheShare: Automated server flash cache space management in a virtualization environment. In *USENIX*, 2014.

[76] B. Milani and N. Navimipour. A comprehensive review of the data replication techniques in the cloud environments: Major trends and future directions. *Journal of Network and Computer Applications*, 64(1):229–238, 2016.

[77] C. Mohan. Tutorial: Application servers and associated technologies. In *VLDB*, 2002.

[78] M.Tikir and J. Hollingsworth. NUMA-aware Java heaps for server applications. In *IPDPS*, 2005.

[79] M. Naftalin and P. Wadler. Java generics and collections. 2006.

[80] S. Noll, J. Teubner, N. May, and A. Boehm. Accelerating concurrent workloads with CPU cache partitioning. In *ICDE*, 2018.

[81] D. Nukarapu, B. Tang, L. Wang, and S. Lu. Data replication in data intensive scientific applications with performance guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 22(8):1299–1306, 2011.

[82] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[83] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 2019.

[84] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[85] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *ASPLOS*, pages 205–216, 1998.

[86] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, 2005.

[87] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, 2012.

[88] J. Philipp, N. De Palma, F. Boyer, and O. Gruber. Self adapting service level in Java enterprise edition. In *Middleware*, 2009.

[89] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Middleware*, 2004.

[90] D. Porobic, E. Liarou, P. Tozun, and A. Ailamaki. ATraPos: Adaptive transaction processing on hardware islands. In *ICDE*, 2014.

[91] A. Quamar, K. Ashwin Kumar, and A. Deshpande. SWORD: Scalable workload-aware data placement for transactional workloads. In *EDBT*, 2013.

[92] T. Rabl and H. Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In *SIGMOD*, 2017.

[93] S. Rajasekaran, S. Duan, W. Zhang, and T. Wood. Multi-cache: Dynamic, efficient partitioning for multi-tier caches in consolidated VM environments. In *IEEE International Conference on Cloud Engineering*, 2016.

[94] J. Rao, X. Bu, C. Xu, L. Wang, and G. Yin. Vconf: a reinforcement learning approach to virtual machines auto configuration. In *ICAC*, 2009.

[95] K. Risvik, Y. Aasheim, and M. Lidal. Multi-tier architecture for web search engines. In *LA-WEB*, 2003.

[96] R.Kingsy and G. Manimegalaib. Dynamic replica placement and selection strategies in data grids- a comprehensive survey. *Journal of Parallel and Distributed Computing*, 74(2):2099–2108, 2014.

[97] U. Rohm, K. Bohm, and H. Schek. Cache-aware query routing in a cluster of databases. In *ICDE*, pages 641–650, 2001.

[98] N. Schiper, F. Pedone, and R. Renesse. The energy efficiency of database replication protocols. In *DSN*, 2014.

[99] H. Schuldt. Multi-tier architecture. In *Encyclopedia of Database Systems*, pages 1862–1865. Springer US, Boston, MA, 2009.

[100] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB*, 7(12):1035–1046, 2014.

[101] M. Serafini, R. Taft, A. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. In *PVLDB*, 2016.

[102] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. An autonomic approach for replication of internet-based services. In *SRDS*, 2008.

[103] S. Sharify, A. W. Lu, J. Chen, A. Bhattacharyya, Ai B. Hashemi, N. Koudas, and C. Amza. An improved dynamic vertical partitioning technique for semi-structured data. In *ISPASS*, 2019.

[104] Y. Shin, M. Ghosh, and I. Gupta. Parqua: Online reconfigurations in virtual ring-based NoSQL systems. In *ICAC*, 2015.

[105] K. Shvachko, . Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *MSST*, 2010.

[106] A. Soror, U. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, 2008.

[107] S. Sotiriadis, N. Bessis, C. Amza, and R. Buyya. Elastic load balancing for dynamic virtual machine reconfiguration based on vertical and horizontal scaling. *IEEE Transactions on services computing*, 12(2):319–334, 2019.

[108] G. Soundararajan, J. Chen, M. Sharaf, and C. Amza. Dynamic partitioning of the cache hierarchy in shared data centers. In *PVLDB*, pages 635–646, 2008.

[109] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *SoCC*, 2015.

[110] A. Stiemer, I. Fetai, and H. Schuldt. Analyzing the performance of data replication and data partitioning in the cloud: the BEOWULF approach. In *ICDE*, 2016.

[111] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[112] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Frans Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transaction on Networking*, 11(1):17–32, 2003.

[113] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-Store: Fine-grained elastic partitioning for distributed transaction processing systems. In *VLDB*, 2014.

[114] D. Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[115] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[116] G. Tesauro. Online resource allocation using decompositional reinforcement learning. In *AAAI*, 2005.

[117] G. Tesauro, N. Jong, R. Das, and M. Bennani. On the use of hybrid reinforcement learning for autonomic resource allocation. In *CLUSTER*, 2007.

[118] J. Tirado, D. Higuero, F. Isaila, and J. Carretero. Multi-model prediction for enhancing content locality in elastic server infrastructures. In *HiPC*, 2011.

[119] J. Tirando, D. Higuero, F. Isaila, , and J. Carretero. Predictive data grouping and placement for cloud-based elastic server infrastructures. In *CCGRID*, 2011.

[120] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst*, 3(1):1–39, 2008.

[121] D. VanderMeer, K. Dutta, and A. Datta. A cost-based database request distribution technique for online e-commerce applications. *MIS Quarterly*, 36(2):479–507, 2012.

[122] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. In *CIDR*, 2011.

[123] X. Wei, S. Shen, R. Chen, and H. Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *USENIX ATC*, 2017.

[124] Jinho Hwang Wei Zhang, Timothy Wood. Netkv: Scalable, self-managing, load balancing as a network function. In *ICAC*, 2016.

[125] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, 2015.

[126] X. Zhang, M. Barrientos, J. Chen, and M. Seltzer. HACC: An architecture for cluster-based web servers. In *USENIX NT Symposium*, 1999.

Acronyms

LB	Load Balancer
AS	Application Server
LC	Local Cache
DB	Database
CC	Cooperative Cache
VM	Virtual Machine
JVM	Java Virtual Machine
UUID	Universally Unique Identifier
NAS	Network Attached Storage
OLTP	Online Transnational Processing
OLAP	Online Analytical Processing
HDFS	Hadoop File System