Performance Evaluation of Web Archiving Through In-Memory Page Cache

Saket Vishwasrao

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Engineering

Edward A. Fox, Chair
Zhiwu Xie
Thomas Hou

April 27, 2017
Blacksburg, Virginia 24061

Keywords: Information Retrieval, Transactional Web Archiving, Caching, Benchmarking, Wikipedia

Performance Evaluation of Web Archiving Through In-Memory Page Cache

Saket Vishwasrao

Abstract

This study proposes and evaluates a new method for Web archiving. We leverage the caching infrastructure in Web servers for archiving. Redis is used as the page cache and its persistence mechanism is exploited for archiving. We experimentally evaluate the performance of our archival technique using the Greek version of Wikipedia deployed on Amazon cloud infrastructure. We show that there is a slight increase in latencies of the rendered pages due to archiving. Though the server performance is comparable at larger page cache sizes, the maximum throughput the server can handle decreases significantly at lower cache sizes due to more disk write operations as a result of archiving. Since pages are dynamically rendered and the technology stack of Wikipedia is extensively used in a number of Web applications, our results should have broad impact.

Performance Evaluation of Web Archiving Through In-Memory Page Cache

Saket Vishwasrao

General Audience Abstract

This study proposes and evaluates a new method for Web archiving. To reduce response time for serving webpages, Web Servers store recently rendered pages in memory. This process is known as caching. We modify this caching mechanism of Web Servers for archival. We then experimentally evaluate the impact of our archival technique on Web Servers. We observe that the time to render a Web page increases slightly as long as the Web Server is under moderate load. Through our experiments, we establish limits on the maximum requests a Web Server can handle without increasing the response time. We ensure our experiments are conducted on Web Servers using technologies that are widely used today. Thus our results should have broad impact.

Acknowledgments

I thank my advisors, Dr. Xie and Dr. Fox for their constant support, guidance and constructive criticism that has helped me stay focused and give my best efforts. I dedicate this work to my parents for their undying love and faith in my abilities. This work would not have been possible without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Web as a Form of Expression

Over the centuries, mankind has found different means for expression. Architecture, paintings, and manuscripts have been used since ancient times. With the advent of new technologies – such as the printing press – books, journals, and conference papers became widely used means for expressing new ideas. Preservation of these forms of expressions has enabled us to understand our past as well as predict the future. It is through this medium that knowledge, art, morals, laws, and customs are propagated though generations. Our ability to understand this preserved information, utilize it in daily life, further improve it, and pass it on to future generations has enabled us to make rapid progress across centuries.

The World Wide Web (WWW or Web) was invented towards the end of the 20th century. Within a short span of time, the Web has become an integral part of modern society. The ease with which information can be recorded and propagated irrespective of geographical distances has caused the Web to become the primary medium for expression today. Blogs, online publications, and social media currently are some of the most common forms of expression on the Web. Currently, about 47% of the people in the world are Web users [ITU (2016)]. It is estimated that there are a billion websites, and the number is rapidly growing [ISC (2016)]. Many characteristics of society are reflected on the Web.

However, much of what appears on the Web is ephemeral or short lived, as is discussed in Section 1.3. It is necessary to continuously archive the Web content, otherwise a lot of information available on the Web would be lost. For this reason, Web preservation has become a necessity.

1.2 Web Archiving

Web Archiving is the process of collecting the resources available on the Web, storing and making them accessible for further research. Multiple organizations are actively pursuing Web archiving. The Internet Archive[1] is one example. Commercial Web archiving software and services are also available to organizations who need to archive their own Web content for heritage, regulatory, or legal purposes.

Similar to any information management technique, Web archiving involves selecting the content to archive, acquisition, organization, storage and retrieval. On account of the sheer size of the Web, it is impractical to archive all of the content. Accordingly, content is often archived based on well defined criteria and human judgment. Software programs that systematically browse the Web, called crawlers, are used to gather such content. The data collected by the crawlers can then be stored in an ordered fashion on huge data stores. This data is further indexed for efficient retrieval, supporting access by services like the WayBack Machine[2].

Archiving techniques for documents, in general, have evolved over a long time. These can be applied to efficiently preserve a wide range of physical publications by libraries. Over the years, different types of publications have been added, but their inherent nature has remained the same. Though in many ways webpages are similar to such documents, there are some significant differences. This presents a new set of challenges for archiving the Web. To understand the motivations as well as to develop efficient methods for archiving, it is important to realize how the Web differs from existing publications.

1.3 The Nature of the Web

The Web shows some unique characteristics that motivate us to think beyond the traditional means for preserving documents.

First, the Web is ephemeral in nature. Websites are constantly evolving and changing. Studies show that the half life of a webpage (period during which half of the webpages disappear) is around 2 years [Koehler (2004); Spinellis (2003)]. These studies only consider the existence of the URLs. Cho and Garcia-Molina (1999) verified that if change in content is taken into consideration, the half life is a mere 50 days. Second, rendered webpages are user specific. The webpages presented may differ depending on the browser used to access the content, the geographic location of the user, etc. As a result, Web archiving can archive only certain instantiations of the webpages, with a degree of variation possible. This is in contrast to publications that are consistent across all users.

[1] https://archive.org/index.php
[2] https://archive.org/web/

Third, to broaden utility, it is necessary to remove the dependence of webpages from specific server technologies. This is to ensure that the archived content is resilient to the Web's evolution and failures. A physical document on the other hand does not depend on any factor once it is created. Thus it can be archived without any resiliency issues.

As we see next, the lack of understanding of these fundamental differences has resulted in added skepticism regarding the need of Web archiving.

1.4 Perspectives on Web Archiving

Preserving the Web is questioned and not widely accepted by all. In this section, we summarise various arguments discussed by Masanés (2006) against the necessity of archiving the Web.

Firstly, the Web is assumed to be a self preserving medium. It is believed that resources deserving to be preserved will be maintained on servers; others will disappear at the creator's will. However there are some fundamental reasons that prevent the Web from being a self preserving medium. For instance, the short term nature of domain name rentals, changes in Web Server technologies, organizational changes, and the need to host servers 24x7 as opposed to the one off nature of printing, all result in constantly changing websites. One might argue that there are content management systems for publishing resources, but these come and go, and related agreements and systems software often change. Often, the tasks of publishing and preserving content are segregated. Publishing, which usually implies creating new content, provides little incentives for preserving said content. Preservation, when it occurs, is driven by a different set of organizations (sometimes called "memory institutions"[3]) based on different goals as compared to publishing.

The second important question is: Is the Web good enough for archiving? This arises from the lack of understanding of Web archiving approaches. Historically, archiving has been governed by factors such as costs of storage, transport, and handling. This has resulted in filtering techniques to increase the scope and quality of archived content while minimizing utilization of various resources. Such techniques are difficult to apply on a document by document basis for the Web due to the sheer size of it, so this is not the best approach. A lot of content on the Web is not accessed by humans at all and thus there is no need to archive it. Boufkhad and Viennot (2003) showed that only 5% of the pages of an large academic website are actually accessed by crawlers for indexing. The distribution of user access to online content follows a power law and thus exhibits a very long tail. Access patterns can be used to determine a suitable depth of crawl for each site. For instance, the Internet Archive uses access patterns to evaluate the depth of crawl for each site [Mohr et al. (2004)]. Evaluating the quality of content archived can be left up to specific organizations or individuals themselves. Wikipedia is a great example of how the content quality of openly

[3] http://internetmemory.org/en/

editable documents is maintained by collaboration.

Third, there are various issues that result in added skepticism to Web archiving. These include privacy concerns, intellectual property rights issues, and copyright obstacles. To address these concerns, some organizations only archive content that is primarily non-commercial or public. On the other hand, much critical information is not publicly available, and indeed is protected by various authentication techniques. For example, access to particular webpage content can be restricted by using a robot.txt file that prevents crawlers from accessing the copyright protected content; such decisions are assumed to be the responsibility of the website owner. Accordingly, privacy and intellectual property rights related issues can be addressed, and Web archiving can easily be limited to the vast amount of readily accessible content for which preservation concerns are not an issue.

Thus we see that the arguments against Web archiving arise mainly from a lack of understanding of the scope and value of the archiving process. Organizations like libraries and museums have played an integral part in preserving artifacts and documents using methods that have evolved over time. Although these methods can be applied to Web archiving, the nature and characteristics of the Web compel us to rethink and adapt new preservation practices. We look at some of the methods for Web archiving in the next section.

1.5 Methods for Web Archiving

There are three primary methods for archiving the Web [Masanés (2006)]. The first is client side archiving. Crawlers crawl the Web to mine webpages and create a copy of them. Second is server side archiving where the website owners create a snapshot of the website data on the server. Third is transactional archiving, which archives every HTTP transaction that happens between the client and the server. We look at each of these techniques in detail.

1.5.1 Client Side Archiving

This is one of the most widely used method for archiving. Crawlers are software programs that start at certain seed pages, parse them, extract links, and fetch the pages pointed to by the links. As long as there are unvisited links to explore, a crawler would explore them and archive the pages. This allows archiving of every page freely available on the Web as long as the crawler gets appropriate authorization. Moreover, a crawler mimics the interaction of users with the Web Server in terms of the requests made. Since crawlers are programs outside of the Web Server, they behave as would a client, hence the term client side Web archiving.

Most crawling technology was built for indexing with application in search engines. In order to use this technology for Web archiving, several factors have to be taken into considerations.

First, crawlers for archiving purposes have to capture the content and not just the links to it. Crawlers used in search engines often store just the links to different content, along with entries in their indexes. The actual content is served by the original server. Copying the resources thus adds significant overhead in terms of bandwidth and time needed for crawling websites. Secondly, not all files can be indexed. Many webpages are unlinked, i.e., the only way to access them is to know their URLs. Since they cannot be reached through a sequence of hyper-links, a regular crawler would simply ignore files that cannot be indexed.

Thirdly, crawlers have to respect the politeness policy of a website. This means there has to be a fixed amount of delay between two requests to the same website. Thus, crawling might require hours and sometimes even days when archiving large websites. Certain pages may change during this time. Thus, though client side archiving ensures that all the content accessible by crawlers is archived, it may not necessary guarantee a consistent single point in time archive of a website [Denev et al. (2011)].

1.5.2 Server Side Archiving

In this type of archiving, files are copied directly from the server into the archives without accessing the HTTP interface. Although the method is extremely simple, it raises serious issues.

For dynamically generated resources, one approach is for the entire server infrastructure be recreated at the archives, as content is generated on the fly. However, archives would then have to have and maintain a growing number of changing infrastructure instances. Alternatively, dynamic content can be archived as static HTML files, but then there might be an enormous amount of space required, making such server side archiving difficult. Many rendered pages access resources from an external Web Server. To recreate the exact webpage using the copy of server infrastructure is not possible in such a scenario as it would be necessary to archive the external servers too.

However, server side archiving is beneficial especially when pages (e.g., forms) that require complex interaction between client and server are to be archived, as it would be difficult to capture this information using crawlers.

1.5.3 Transactional Archiving

Transactional Archiving consists of capturing and archiving all materially distinct responses produced by a website, regardless of their content type and how they are produced [Fitch (2009)]. Transactional Web archiving happens at the gateway of the origin server and archives the HTTP responses to user requests, typically in real time, when the HTTP transactions occur. This archiving technique is independent of the content type and the method of response generation. Since the HTTP responses are archived and not the raw data, we

do not need the original execution environment for recreation of responses. Systems like PageVault [Fitch (2009)] and SiteStory [Van de Sompel (2012)] implement such a method.

This method has an advantage due to the fact that as long as the content is viewed by at least one user, it would be archived. Pages that are never accessed (the hidden Web) would not be archived. As long as there is at least a slight difference in the requests/responses, the content is archived. Further research has to be done in order to adapt this approach to the numerous ways of personalized and location specific responses.

We discussed above three different methods for Web archiving. Crawling technology is by far the most widely used for archiving. However, as discussed earlier there are temporal inconsistencies in the resulting archived content. To make server side and transactional archival techniques more practical and increase their use in archival, further research is needed. Since both can be used only with the cooperation of the website owners, it is necessary to study the effect they have on websites in terms of performance and the incentives they offer to the website owners. In this work, we look at ways to further improve these archiving techniques, and evaluate their performance.

1.6 Improving Web Archiving Techniques

Web archiving is not a core server functionality. To implement Web archiving on the server, it is necessary to ensure that its performance impact is tolerable. Website owners do not want users to have a negative experience due to higher latencies. Anything that utilizes server resources like CPU, memory, and network bandwidth adds to the costs of running the server. Thus, resources needed for archival functionalities should be minimal. In fact, the added cost should be justifiable in terms of the benefits provided by archiving.

Recent transactional archiving tools like Sitestory [Van de Sompel (2012)] typically add modules to the Web Server like Apache to archive all the HTTP requests. A performance evaluation [Brunelle et al. (2013)] using static webpages shows slight increase in Web Server latencies. The impact on Web Server capacity has not been clearly shown though. Also, as modern Web infrastructures render pages dynamically, extra tiers (e.g., database) are added to the Web Server stack. These extra tiers may slow down the Web Server and possibly magnify the impact on a Web Server.

Evaluation of the impact of archiving techniques on a realistic Web setup is needed. Techniques that possibly suggest performance improvements in latency, and maximum request handling capacity of the server while archiving, should be explored.

1.7 Archiving using In-memory Cache

A Web Server uses a page cache to store frequently accessed pages in memory to reduce response time. Our system uses this in-memory caching infrastructure of the Web Servers to enable archiving. Rather than archiving every response from the server as in transactional archiving, we archive data whenever the page cache is updated. This implies that when a new page is written to the cache, it is written to the archive as well. The frequency at which pages are written to the cache (cache misses) should determine the effect of archiving on server performance.

Figure 1.1 shows our system architecture we built for testing this technique. We use Nginx as our Web Server, Redis as page cache and object cache, Mediawiki for rendering dynamic pages using PHP, and MySQL as our database. The reasons for selecting each of these components are explained in detail in Chapter 3.

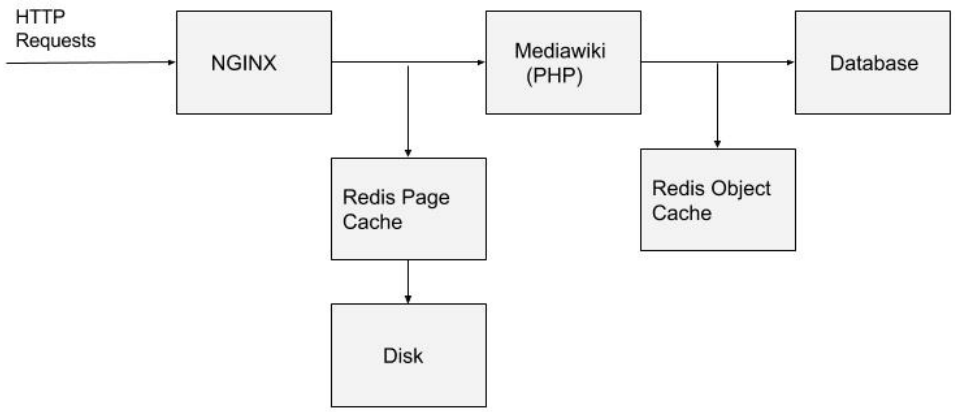


Figure 1.1: System Architecture

1.8 Contributions

We extend the idea of transactional archiving and propose a new approach. We then experimentally evaluate our technique using a realistic workload and study its effect on server performance.

Specifically, in this work we look to answer the following questions:

- Is it feasible to archive every update operation of the page cache?
- If so, how many simultaneous requests can the server handle while archiving as compared to not archiving?
- How does archiving affect the latencies in rendering pages?
- How does varying the maximum cache memory affect the server performance?
- What is an ideal cache size, if any, for archiving and what is the tradeoff?

Our hypothesis is that since the cache is not updated at every request to the Web Server, archiving happens at a lower frequency. This should reduce the degradation in server performance due to archiving.

To ensure that our system is tested under realistic workload and access patterns, we run our experiments on the Greek version of Wikipedia which contains 119440 pages and around a 475,000 page revision history. We use a read trace consisting of 10704 of the most frequently accessed URLs. The pages are rendered dynamically. We use Redis as the in-memory page cache and its persistence mechanism is leveraged for archival purposes. A Solid State Drive (SSD) is used to store the archives. We use Amazon Web Services (AWS) cloud infrastructure to run our experiments. This way, the system and experiments are easily replicable for future research.

The size of the page cache is dependent on the hardware resources available to the website owner. We show that latencies are not significantly affected due to archiving using our system, as long as the server has not reached its maximum capacity. We also find that the maximum server capacity degrades significantly at lower cache sizes. This happens because at small cache sizes, as the cache gets updated frequently, more content is getting archived and thus there are more reads/writes to be performed. However, we find that the server capacity while archiving does not degrade a lot as compared to not archiving for higher cache sizes. At higher cache sizes, since there are lower cache misses, the cache is updated less frequently. Thus, less content is archived resulting in a lower overhead on the servers.

Thus, we propose a new approach to Web archiving and study its performance impact on the Web Servers using a realistic workload. We find that as long as the website owners are willing to operate the servers below the maximum shown capacity, our system can be practically used for archiving without significant increase in latencies.

1.9 Outline of the Thesis

In this chapter, we discussed the idea of Web archiving, along with some of the issues and methods related to it.

We review the literature and discuss the ongoing research effort in Web archiving in Chapter 2. Chapter 3 discusses the methodologies we use for experimentation. We give a detailed description of the design considerations and various caveats upon which our experiments are based. We discuss the hardware architecture, the software stack used, and the techniques involved in optimizing them. Chapter 4 discusses the results obtained. We provide various metrics to show the effect of archiving using in-memory cache on Web Servers. Chapter 5 discusses the insights and conclusions we derive out of our experiments.

Chapter 2

Review of Literature

2.1 Web Archiving

Web archiving refers to the collection and long-term preservation of data available on the Web [Masanés (2006)]. Techniques for Web archiving have evolved over the years. Early on, Web archiving was performed mostly by using crawlers. Mercator was the first scalable Web crawler [Heydon and Najork (1999)]. A distributed crawler was later created from Mercator by Shkapenyuk and Suel (2002). The Internet Archive utilized this crawling technology for archiving. Heritrix is an archival-quality and modular open source crawler, developed at the Internet Archive [Mohr et al. (2004)]. They also created the Wayback Machine[1] which provides tools to search and access the content. There have been numerous other efforts to create archives of the Web by using crawling technology, including in different countries like Australia [Phillips (2009)], United Kingdom [Bailey and Thompson (2006)], and Sweden [Arvidson and Lettenstrm (1998)].

Use of crawling technologies has issues related to data consistency. The evolving nature of the Web causes the webpages to change constantly [Ntoulas et al. (2004); Fetterly et al. (2003)]. Many studies have been performed to determine how the nature of webpages change. Olston and Pandey (2008) provide an empirical analysis for the longevity measurements of webpages. To understand how the Web evolves over time, Cho and Garcia-Molina (1999) conducted a comprehensive experiment on 720,000 webpages from 270 websites over 4 months. They found that the half life for average webpages is 50 days, and 40% of the webpages change within a week. A number of factors result in changes to a webpage such as domain level and document size [Fetterly et al. (2003) Brewington and Cybenko (2000)]. Since archiving can take a long time due to the size of the website, it is difficult to create a single point of time snapshot of the website [Spaniol et al. (2009)]. Denev et al. (2011) propose a framework to assess archiving quality, and provide strategies for effective crawling.

[1] https://archive.org/web/

To avoid temporal differences and data inconsistencies during archiving, Fitch (2009) proposed a new technique known as transactional archiving. This involves archiving all response-request pairs regardless of content type and how they are produced. This idea was implemented by the Page Vault system, which consists of four components. The filter module runs as a server plugin identifying potentially different request-response pairs. As the module does not have a global view (Web servers can be distributed) and maintains only the recent local history, there can be some duplication of responses. The distributor module, run as a separate process, parses through the stored content by the filter and further removes the duplicates. This information is then archived in a persistent storage by the archiver which is usually run on a separate machine from the Web server. The archived data can be then accessed by a query servlet which is a Java based application.

Los Alamos National Laboratory developed SiteStory [Van de Sompel (2012)], an open-source transactional Web archiving system. It consists of a "mod_sitestory" module that sends the response generated by the Web server to the SiteStory archive. The archived responses can be accessed using a Memento-based protocol from the SiteStory Archive [de Sompel et al. (2009)]. To send HTTP responses to the SiteStory Archive, the Web server has to perform additional tasks. Brunelle et al. (2013) studied the impact of using SiteStory on server performance.

A survey of all the Web archiving initiatives is provided by Gomes et al. (2011). There have been at least 42 Web archiving initiatives to date. Most of these initiatives are highly selective regarding the content they archive. 80% of these archives hold content related to the hosting country or institution. There are at least three organizations that provide Web archiving services. The total amount of archived data is at least 6.6 PB (181,978 million webpages). Accordingly, efficient search mechanisms are required to enable access to this information, which raises new technological challenges. The largest Web search engine indexes only 20% of this data. Gomes et al. (2011) showed that 89% of the initiatives surveyed support access to the history of a given URL, 79% enable searching metadata, 67% provide full-text search over archived contents, and 50% provide full online access to search mechanisms and archived content. About 62% of these initiatives are based on the WayBack Machine.

2.2 Caching Solutions

As discussed in Section 1.7, we leverage the in-memory caching infrastructure for archiving. Thus, it is important to discuss various caching solutions that are widely used in Web servers.

We next look at most frequently used caches on Web server's and evaluate the feasibility of implementing our archival approach using them.

Varnish

Varnish Cache[2] is a Web application accelerator also known as a caching HTTP reverse proxy. It can be installed in front of any server that speaks HTTP, and configured to cache the contents. Varnish Cache is very fast. It typically speeds up delivery by a factor of 300 - 1000x, depending on the architecture. Varnish uses the Varnish Configuration Language (VCL) for configurations.

In order to use Varnish for our implementation, a new module implementing the desired functionality could be created. This module then has to be compiled and installed on the server running Varnish.

Memcached

Memcached[3] is a free, open source, high-performance, distributed memory object caching system. It also can be used as an in memory key value store for storing chunks of arbitrary data. Memcached is simple yet powerful. Its simple design promotes quick deployment, ease of development, and solves many problems facing large data caches. Its API is available for most popular languages. Similar to Varnish, modules can also be created for Memcached for implementing the archival process.

Redis

Redis[4] is an open source (BSD licensed) in memory data structure store which can be used as a database, cache, and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, etc. Redis has built-in replication, Lua scripting and different levels of on-disk persistence. It also provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

Our Choice

We needed a Web cache that can be configured or modified to write to disk (archive data). This could be done by using a seperate low priority process that records the cache transactions and writes to disk. It is necessary to build a plugin or leverage an inbuilt feature in an innovative way in order to fulfill our aim. Modifying the source code is not feasible as that would mean recompiling the cache software from scratch. Such a solution would be discouraging since the website owners would not want to rebuild their Web caches on the

[2] https://www.varnish-cache.org/docs/5.1/
[3] https://memcached.org/
[4] https://redis.io/

servers. We do not want to modify the source code as that would lead to recompilation and possibly unexpected effects on the performance of the cache. We needed a plugin or module that could possibly be loaded at runtime for performing our tasks.

Varnish and Memcached, though widely used, do not have any feature that can be leveraged for this functionality. However, Redis turns out to be an ideal candidate for this application. Redis has a built in persistence mechanism which is highly optimized and configurable. The performance of Redis is comparable to that of Memcached, which is widely used in many Web servers[5]. Thus, the performance effect of the cache on the server is similar to that of a real life Web server. We leverage this persistence mechanism for the archiving procedure. The source code does not have to be modified to implement the archiving procedure.

2.3 Summary

In this chapter we discussed some of the early crawling technologies used for Web archiving. We saw that due to the rapidly changing nature of the webpages, creating a single point of time snapshot was difficult using crawling. Transactional archiving techniques came into being to address these issues. However these techniques are still not widely used. We then discussed some of the initiatives all over the world for archiving the Web. Further, since we leverage the caching infrastructure for archiving, we looked into different caching solutions, and described some of the most frequently used caches.

[5]http://oldblog.antirez.com/post/redis-memcached-benchmark.html

Chapter 3

Methodologies

Our goal is to leverage the caching infrastructure of Web servers for archiving, and study the impact of it on server performance as discussed in Section 1.8. In this chapter, we look at the methods to do so. Section 3.1 discusses the motivations behind using Wikipedia as our test infrastructure. We explain our rationale regarding various configurations of different components of the server stack. Section 3.2 gives details on how the AWS cloud infrastructure is utilized as the hardware platform for performing our experiments.

3.1 Wikipedia

Wikipedia is a free encyclopedia, written collaboratively by the people who use it. But its main advantage lies in the ease with which it can be hosted using its open source stack. Mediawiki is a well maintained and widely understood content management system (CMS) written in PHP used for hosting Wikipedia. It is comparable with other PHP based CMSs like Drupal[1] and WordPress [2] in terms of technology stack and performance.

We are interested in evaluating the archival performance of the core server stack. In order to avoid the use of load balancers and multiple caching levels, we do not want to host a very large Wikipedia. We pick the Greek version of Wikipedia which we found to be of the right size. Recent XML dumps of Greek Wikipedia[3] are available and regularly updated. We use the October 2016 dump with the full revision history for our experiments.

We create the server stack as illustrated in Figure 1.1. The choice of each component is influenced by a number of different considerations as illustrated in the following sections.

[1] https://www.drupal.org/
[2] https://wordpress.com/
[3] https://dumps.wikimedia.org/elwiki/

3.1.1 Web Server

Right from early days of the Internet, Apache[4] has been widely used as a Web server. Its simplicity was the key to its success during those early days. Apache uses a one-connection-per-process model. A new process is created for every new HTTP connection request. If all processes are busy, new processes are spawned. It was easy for developers to add code (in form of modules) to Apache at any point of time, because in the event of some resource leakage or a crash, only the process running the faulty code would be effected. The rest of the processes would be isolated. This helped in Apache becoming popular.

However over the years, the size and the number of components needed by the client to render a webpage has increased. Apache's process-per-connection model does not scale well for these high requests and heavy pages (having a lot of embedded scripts). Web clients often greedily use resources by opening simultaneous connections to the server so that multiple parts of the webpage can be downloaded in parallel. This results in creation of large number of processes on the Web server consuming a lot of memory and leads to high context switch time slowing the server.

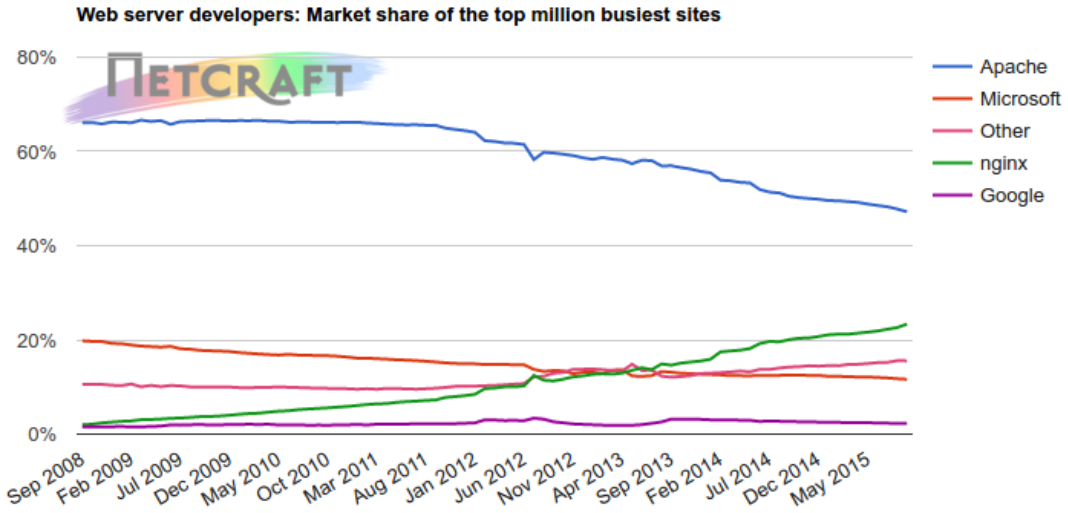


Figure 3.1: Market share of Web servers [Netcraft (2015)]

Nginx[5] was built to address these scalability issues in Apache. It improves the scalability by using an event driven architecture. Every worker process can handle multiple connection requests simultaneously unlike Apache. This comes at a cost of making it difficult to build

[4]https://httpd.apache.org/
[5]https://www.nginx.com/resources/wiki/

modules for Nginx. Developers have to be careful to build efficient error-free code that interacts with the complex event-driven core code to avoid blocking operations.

Nginx does not implement all the functionality to run an application. Rather it depends on third-party servers like PHP-FPM, Node.js, etc. for running backend scripts. It acts as a light weight server that is highly scalable and provides high performance. Over a period of time, market share of Nginx has grown steadily. See Figure 3.1.

Thus Nginx is widely being used. It can be configured to use Redis as the page cache and offers efficient PHP support. Hence we use Nginx as our Web server.

Redis Interface

Nginx does not provide built in Redis support out of the box. However there are open source tools that can be used to interface with Redis, and use it as a page cache. The "srcache" module provides a transparent caching layer for arbitrary nginx locations which can either be an upstream location (Redis in this case) or a static disk file. This module uses the "redis2-nginx-module" for writing data to Redis. This nginx upstream module is used to communicate with Redis in a non-blocking way. It implements a full Redis 2.0 unified protocol including the pipelining support.

Rendering Dynamic PHP content

Nginx supports FastCGI handlers for serving dynamic content scripts such as PHP. FastCGI is protocol used to improve performance by not running each request as a separate process. Unlike Apache which can handle PHP processing directly using the corresponding php-module, Nginx relies on a separate PHP processor to handle PHP requests. This is handled by the PHP-FPM (FastCGI Process Manager) processor.

3.1.2 Page Cache

As discussed in Chapter 2, Redis can be used both as an in-memory cache and a persistent data store. Redis stores data as key value pairs in-memory. Every page accessed by the client is stored in Redis as a key value pair. The relative URL of the resource accessed by the client is stored as the key and the HTML response generated by PHP is stored as the value. It supports multiple eviction policies like least recently used, random selection, etc. In addition, each key has a fixed time to live (TTL). The key is evicted after the TTL expires. We use Redis as a least recently used (LRU) page cache.

The biggest advantage of using Redis as page cache is its persistence mechanism. This feature was added to Redis in order to ensure that the data stored is not lost during failure.

Recollect that Redis is a single threaded application. To implement persistence, Redis forks a low priority child process that writes stored data to the disk.

Redis persistence is highly configurable. There are two different ways in which this feature can be used.

- Redis Database File (RDS)

  Redis creates a point in time snapshot periodically. The interval between successive snapshots can be configured in the Redis configuration file.

- Append Only File (AOF)

  Redis appends any operation that modifies the data stored in-memory to an AOF file on the disk. Thus this file is a record of transactions that has happened in Redis.

The Redis persistence mechanism was built to offer a reliable backup. Usually successive RDS files are overwritten. Similarly, if the AOF file gets too big, it is overwritten by the current cache contents. This is because, for backup, content evicted from the cache is no longer needed. However, looking from the perspective of archiving, we want to record all changes made to the data, but not overwrite them. Fortunately, Redis can be configured to disable this overwriting feature.

RDS provides a snapshot of cache content. We do not want the state of the page cache at a particular time. In order to test our hypothesis, we want to record all the transactions that involve the cache. The AOF mechanism offers precisely this function. Hence, we use the AOF persistence mechanism of Redis for archiving the cache content.

In our experiments, any modification to the data in Redis is recorded in the AOF file. As the page cache stores all the rendered HTML content and any modified or evicted data is recorded, the AOF file represents our archive of the website. The process of separating unique individual webpages is left for future research. This task should not be difficult as the pages (values) are well identified by their URLs (keys).

The AOF mechanism can be further fine tuned. The child process that executes this mechanism writes every cache update operation to the in-memory kernel buffers. The actual transfer of data from the kernel buffers to the disk happens when an 'fsync()' system call is called by the Redis child. It is not the data transfer to the kernel buffers, but the system call which is expensive and has high impact on the cache performance.

Redis offers the following three options to configure when this system call is executed.

- Execute fsync() on every update operation
- Execute fsync() every second

- No deadlines to execute fsync(). It is usually executed every 30 seconds which is the default Linux kernel configuration.

We conduct our experiments by setting no deadlines on the 'fsync()' system call since we want to minimize the impact of persistence on cache performance. Under high load, the kernel buffer may overflow before the data is written to the disk by the system call. This may lead to some of the update operations not getting logged to the AOF file leading to loss of some pages.

3.1.3 Mediawiki

Mediawiki[6] is a powerful, scalable system and a feature-rich wiki implementation that uses PHP to process and display data stored in a database, such as MySQL. Mediawiki dynamically renders HTML wiki pages as requested by the client. We consider a few design considerations for conducting our experiments in order to reduce the rendering time of each page. Figure 3.2 and 3.3 compares the pages rendered by our server as compared to an actual page retrieved from the live Wikipedia Server.[7]

External Resources

A lot of resources are usually provided by external repositories that host frequently used content. We disable the access to such repositories. For instance, access to the Wikipedia Commons repository, which hosts a lot of images, video, and sound content, is disabled. The embedded script accesses 'WikidataFR' which accesses the Wikidata[8] repository for information. Since we disable it, raw script is shown on the rendered webpage. See point 5 in Figure 3.2.

Non PHP Scripting within Pages

We disable all embedded scripts that need resources outside of the database for rendering pages. This is to avoid any external factors to effect our experiments. For instance, the Scribunto extension which allows embedding scripting languages like Lua in webpages, is disabled, since we found that they often also need resources outside of the database. See marking 1,2 in Figure 3.2. We find that raw scripts like '#invoke' are present in the rendered pages as a result of this.

[6] https://www.mediawiki.org/wiki/MediaWiki
[7] https://el.wikipedia.org/wiki/ Retrieved 17 April 2017
[8] https://www.wikidata.org/wiki/Wikidata:Main_Page

Images and Thumbnails

We do not populate the database with all the images necessary for the pages. When an image is not found, Mediawiki renders a new "404 not found" page for each such failed request and links the original page to this new page. For pages having a large number of images, every failed image request results in a new not found page being generated. This results in severe performance degradation. To avoid such multiple page creations, Nginx is configured to redirect to a static page for each image request. Image thumbnails are also hyper-linked to the same page. Generating thumbnails on the fly while rendering pages is disabled. This reduces the page rendering time as new not found pages are not created if an image is not found. See marking 2 in Figure 3.2.

Page Formatting

We use the standard page formatting and skins with necessary extensions to ensure that the rendered page is similar to that of the real Greek Wikipedia. See marking 3 in Figure 3.2. The page theme, skins, and the formatting are similar to the actually rendered page in Figure 3.3.

Δημιουργία λογαριασμού Σύνδεση

Set $wgLogo to the URL path to your own logo image.

Σελίδα Συζήτηση Ανάγνωση Επεξεργασία Προβολή ιστορικού Αναζήτηση

Αρχική σελίδα
Πρόσφατες αλλαγές
Τυχαία σελίδα
Βοήθεια

Εργαλεία
Τι συνδέει εδώ
Σχετικές αλλαγές
Ειδικές σελίδες
Εκτυπώσιμη έκδοση
Σταθερός σύνδεσμος
Πληροφορίες σελίδας
Παραθέστε αυτή τη σελίδα

3

Μικρόδασος Κιλκίς

Από Greek Wiki

1

{{#invoke:Coordinates|coord}}{{#coordinates:41|04|09|N|22|34|18|E|type:city_region:GR |primary |name= }}

</th></tr>

Μικρόδασος	
Πόλη	
{{#invoke:Location map/multi|container}}	
Χώρα	Ελλάδα
Περιφέρεια	Κεντρική Μακεδονία
• Δήμαρχος	{{safesubst:#invoke:WikidataFR|formatStatements|property=P6|value=|lang=el}}
Πληθυσμός	400 (2011)
Ταχυδρομικός κώδικας	{{safesubst:#invoke:WikidataFR|formatStatements|property=P281|value=61200|lang=el}}
Τηλεφωνικός κωδικός	{{safesubst:#invoke:WikidataFR|formatStatements|property=P473|value=23430|lang=el}}
Πολιούχος	{{safesubst:#invoke:WikidataFR|formatStatements|property=P417|value=|lang=el}}
Ιστοσελίδα	{{safesubst:#invoke:WikidataFR|formatStatements|property=P856|value=|lang=el}}

2

5

Το **Μικρό Δάσος** ή **Μικρόδασος** είναι χωριό του δήμου Πολυκάστρου, της επαρχίας Κιλκίς, του νομού Κιλκίς, κείμενο στη δυτική όχθη του Αξιού.

Οι κάτοικοί του είναι Σαρακατσάνοι και Πόντιοι πρόσφυγες (που αποτελούν και την πλειοψηφία). Οι Πόντιοι κατάγονται από το Καρς, την Τσάλκα τη Σάντα και την Αργυρούπολη. Η παλαιότερη ονομασία του χωριού ήταν Σμόλι. Η σημερινή ονομασία δόθηκε από το μικρό δάσος που βρίσκεται στη βορειοδυτική πλευρά του χωριού. Πριν το 1912 το χωριό ήταν αγρόκτημα Τούρκου μπέη, όπου εργάζονταν χριστιανοί κολίγοι.

Περιεχόμενα [απόκρυψη]

4

Ιστορικά στοιχεία [επεξεργασία]

Την περίοδο των Ελληνιστικών χρόνων κατοικείται ο οικισμός στη θέση Τραπέζι[1], στο Μικρόδασος, κοντά στην όχθη του Αξιού, όπου υπήρχε ναός του Απόλλωνα. Άγαλμα του Απόλλωνα εκτίθεται στο Αρχαιολογικό Μουσείο Κιλκίς, και πρόσφατα ανακαλύφθηκε και η νεκρόπολη του οικισμού.

Παρά την ίδρυση της Βουλγαρικής εξαρχίας το 1870 οι κάτοικοι του Σμολίου παρέμειναν πιστοί στο πατριαρχείο Κωνσταντινουπόλεως. Το 1879 με την εγκατάσταση του Βούλγαρου ομνίτη επισκόπου στο Κιλκίς, Νείλου Ισβόρωφ, οι

Figure 3.2: A Rendered Page by the Server

Χωρίς Σύνδεση Συζήτηση για αυτή την IP Συνεισφορές Δημιουργία λογαριασμού Σύνδεση

Λήμμα Συζήτηση Ανάγνωση Επεξεργασία Επεξεργασία κώδικα Προβολή ιστορικού Αναζήτηση σε Βικιπαίδεια

ΒΙΚΙΠΑΙΔΕΙΑ
Η ελεύθερη εγκυκλοπαίδεια

Κύρια πύλη
Κατάλογος λημμάτων
Προβεβλημένα λήμματα
Τρέχοντα γεγονότα
Τυχαίο λήμμα

Συμμετοχή
Πύλη Κοινότητας
Αγορά
Πρόσφατες αλλαγές
Βοήθεια
Επικοινωνία
Δωρεές

Εργαλεία
Συνδέσεις προς εδώ
Σχετικές αλλαγές
Ειδικές σελίδες
Σταθερός σύνδεσμος
Πληροφορίες σελίδας
Αντικείμενο Wikidata
Παραπομπή

Εκτύπωση/εξαγωγή
Δημιουργία βιβλίου
Κατέβασμα ως PDF
Έκδοση εκτύπωσης

Άλλες γλώσσες
Български
Македонски
Επεξεργασία συνδέσμων

Wikimedia CEE Άνοιξη 2017
Πάρτε μέρος στον διεθνή διαγωνισμό συγγραφής λημμάτων για την Κεντρική και Ανατολική Ευρώπη, εμπλουτίζοντας τη Βικιπαίδεια και κερδίζοντας βραβεία!

Μικρόδασος Κιλκίς

Από τη Βικιπαίδεια, την ελεύθερη εγκυκλοπαίδεια

Συντεταγμένες: 41°04'09"N 22°34'18"E

Το **Μικρό Δάσος** ή **Μικρόδασος** είναι χωριό του δήμου Πολυκάστρου, της επαρχίας Κιλκίς, του νομού Κιλκίς, κείμενο στη δυτική όχθη του Αξιού.

Οι κάτοικοί του είναι Σαρακατσάνοι και Πόντιοι πρόσφυγες (που αποτελούν και την πλειοψηφία). Οι Πόντιοι κατάγονται από το Καρς, την Τσάλκα τη Σάντα και την Αργυρούπολη. Η παλαιότερη ονομασία του χωριού ήταν Σμόλι. Η σημερινή ονομασία δόθηκε από το μικρό δάσος που βρίσκεται στη βορειοδυτική πλευρά του χωριού. Πριν το 1912 το χωριό ήταν αγρόκτημα Τούρκου μπέη, όπου εργάζονταν χριστιανοί κολίγοι.

Πίνακας περιεχομένων [Απόκρυψη]
1 Ιστορικά στοιχεία
2 Αξιοθέατα
3 Αναφορές
4 Παραπομπές

Μικρόδασος
Πόλη
Μικρόδασος

Χώρα	Ελλάδα
Περιφέρεια	Κεντρική Μακεδονία
Πληθυσμός	400 (2011)
Ταχυδρομικός κώδικας	61200
Τηλεφωνικός κωδικός	23430

Ιστορικά στοιχεία [Επεξεργασία | επεξεργασία κώδικα]

Την περίοδο των Ελληνιστικών χρόνων κατοικείται ο οικισμός στη θέση Τραπέζι[1], στο Μικρόδασος, κοντά στην όχθη του Αξιού, όπου υπήρχε ναός του Απόλλωνα. Άγαλμα του Απόλλωνα εκτίθεται στο Αρχαιολογικό Μουσείο Κιλκίς, και πρόσφατα ανακαλύφθηκε και η νεκρόπολη του οικισμού.

Παρά την ίδρυση της Βουλγαρικής εξαρχίας το 1870 οι κάτοικοι του Σμολίου παρέμειναν πιστοί στο πατριαρχείο Κωνσταντινουπόλεως. Το 1879 με την εγκατάσταση του Βούλγαρου ουνίτη επισκόπου στο Κιλκίς, Νείλου Ισβόρωφ, οι κάτοικοι του χωριού ασπάστηκαν την Ουνία. Μετά το 1893, όταν ο νέος Βουλγαρουνίτης επίσκοπος, Λάζαρος Μλαντένωφ προσχώρησε στην Βουλγαρική εξαρχία, οι Σμολιώτες προσχώρησαν σταδιακά και αυτοί στη Βουλγαρική εκκλησία. Μετά την συνθήκη του Νεϊγύ, το 1919, και το Πρωτόκολλο Πολίτη-Καλφώφ του 1924 οι κάτοικοι αποχώρησαν στη Βουλγαρία.[2] Στο χωριό συνήφθησαν δύο από τις σημαντικότερες μάχες του Μακεδονικού Αγώνα στην περιοχή. Στις 13 Ιανουαρίου του 1905 ο οπλαρχηγός Μιχαήλ Σιωνίδης με το σώμα του, κατατρόπωσε τον κομιτατζή Αργύρη και την τσέτα του (σκοτώθηκαν 13 κομιτατζήδες). Την 1η Μαρτίου του ίδιου έτους οι τσέτες του Αποστόλ Πέτκωφ και του Σάββα Μιχαϊλοφ, οι οποίες αριθμούσαν 42 άνδρες, περικυκλώνονται στο Σμόλι από Οθωμανικό στρατό και

Figure 3.3: Actual Greek Wiki Page

Create Account Connection

Set $wgLogo to the URL path to your own logo image.

Page Discussion Reading Processing View history Αναζήτηση

Revision history of page "Mikrodasos Kilkis"

View logs for this page

Home
recent changes
random page
Aid

Tools
What links here
related changes
Atom
special pages
information page

Browsing history

From year (and earlier): 2017 From month (and earlier): all Transition

Dispute selection: Mark the radio boxes of the versions to compare and hit enter or the button at the bottom.
Legend: **(cur)** = difference of the latest version, **(prev)** = difference from the previous version, **m** = minor change.

Compare selected revisions

- (current | previous) 5:54 14 February 2016 Pyraechmes (Talk | contributions) . . (7,506 bytes) (+34) . . (Undo)
- (Current | previous) 7:03, 13 September 2015 Yobot (Talk | contributions) **m** . . (Seven thousand four hundred seventy-two bytes) (-15) . . *(Correction of clerical errors in code using AWB (11457))* (undo)
- (Current | previous) 13:41, January 26, 2015 Postscriptum123 (Talk | contributions) . . (7.487 bytes) (+50) . . (Undo)
- (Current | previous) 12:18 6 September 2014 Yobot (Talk | contributions) **m** . . (7.437 bytes) (-22) . . *(Editorial correction code using AWB (10454))* (undo)
- (Current | previous) 18:19, 30 June 2014 Magioladitis (Talk | contributions) **m** . . (Seven thousand four hundred fifty-nine bytes) (-33) . . *(WPCleaner v1.33b - Fixed using Wikipedia: WikiProject Check Wikipedia (League with the same link text))* (undo)
- (Current | previous) 12:27, January 2014 11 Pyraechmes (Talk | contributions) . . (7,492 bytes) (0) . . (Undo)
- (Current | previous) 19:43, January 2014 8 Pyraechmes (Talk | contributions) . . (7,492 bytes) (+27) . . (Undo)
- (Current | previous) 19:42, January 2014 8 Pyraechmes (Talk | contributions) . . (7.465 bytes) (-12) . . (Undo)
- (Current | previous) 19:16, 8 January 2014 109.242.91.97 (Talk) . . (7.477 bytes) (+42) . . (Undo)
- (Current | previous) 12:02, April 1 2013 EmausBot (Talk | contributions) **m** . . (7,435 bytes) (-33) . . *(Bot: Migrating 2 langlinks, now Provided by wikidata on d: Q3401633)* (undo)
- (Current | previous) 11:42, January 20, 2013 MerlIwBot (Talk | contributions) **m** . . (7.468 bytes) (+16) . . *(Robot: Adding: mk: Смол)* (undo)
- (Current | previous) 17:10 24 September 2012 Pyraechmes (Talk | contributions) . . (7.452 bytes) (-4) . . (Undo)
- (Current | previous) 17:08 24 September 2012 Pyraechmes (Talk | contributions) . . (7,456 bytes) (+350) . . (Undo)

Figure 3.4: Revision History of a Rendered Page by the Server

Figure 3.5: Revision History of Actual Page

More Optimisations

We further enable the "Miser mode" and disable page hit counters to prevent expensive database update operations. Mediawiki system messages are modified to reduce parsing time. See Appendix A for more details on configuration settings for Mediawiki.

Figure 3.4 and 3.5 shows that revision history[9] of the rendered and the actual pages. Since we load the entire revision history for each page, we see that it is the same for both the pages. We ensure that the core text in a webpage is unaffected and we get a close approximation of the actual rendered page albeit eliminating all the external resources needed. This may result in raw scripts being present on the HTML pages, but it eliminates the time needed to access resources on external infrastructure. Thus we also remove the effect of external services on our experiments. Only the components shown in Figure 1.1 have an effect on the rendering time and the Web server performance.

3.1.4 Object Cache

To reduce the rendering time of pages, Mediawiki uses an extra caching layer. This caching layer is used to store frequently used PHP objects and database queries. This could also include results of a highly time consuming piece of code. The cache used to store this metadata is called as the object cache. Traditionally, Memcached and PHP object caches like APC are used to boost performance by Mediawiki. These caches have built in support. In addition to these, custom cache objects can be defined to use new caching solutions.

Mediawiki uses object caches for the following purposes. Each can be enabled in the local settings file.

- Main Cache: Store computed results of time consuming scripts, database queries, etc.
- Session Cache: Used to store session information for a user
- Message Cache: Store Mediawiki messages needed across different scripts
- Parser Cache: Used to store parsed pages. Pages can be served from this cache for a different user with similar settings.

We use a separate Redis instance as the cache for all the above resources. See Figure 1.1. This Redis instance is hosted on a different socket than that of the instance serving as the page cache.

To handle multiple requests, Redis uses multiplexing and non-blocking I/O. Redis uses a single thread architecture. The Redis thread queues the received requests in an internal data structure. This process is called pipelining. Thus multiple requests are queued, even if old requests are not executed; hence the client does not have to wait for replies.

[9]Translated to English with Google Translate for readability

Caches like Memcached follow a multi-thread model. To serve multiple requests, Memcached creates multiple threads. To prevent these multiple threads from accessing the same data at the same time, Mediawiki uses a locking mechanism similar to a mutex to maintain data integrity. The thread performing a write operation has to acquire a lock on any resource being edited. No other thread can modify the resource until the lock is released.

To best leverage the single threaded, non-blocking nature of Redis, slight modifications have to be made to the Mediawiki source code. Locks are no longer needed to ensure data consistency, since instructions are pipelined and executed in an atomic fashion. There is no easy way to disable these locks. For non-blocking caches, its best to set the lock timeout to 0 in the source code. This way, even though locks are set, they expire (time out) immediately, which is equivalent to disabling locks.[10]

3.1.5 Database

MySQL is used as the database. Mediawiki has comprehensive support for exporting XML dumps to MySQL as well as for accessing the information already present for rendering pages.

3.1.6 Importing the Wiki Dump

There are several ways to import commonly available Wikipedia dumps[11] into the database. "mwdumper"[12] is a Java based application that can be used to read, write, and convert Mediawiki XML dumps. However we found that, though this is a faster way of importing the dump (10000 revisions/sec), there are consistency issues with the data written to the database. We find that there are errors in rendering certain pages due to corrupted database entries. This results in errors in the server even when the server is not overloaded.

Another way is to use the default PHP script provided with Mediawiki. Though the import is slower (1000 revisions/sec), the data imported is consistent and we get better server performance. Thus we use this script for importing dumps for our experiments[13].

[10] https://github.com/VTUL/Archiving-with-InMemory-Cache/tree/master/Configurations/Mediawiki
[11] https://dumps.wikimedia.org/elwiki/
[12] https://www.mediawiki.org/wiki/Manual:MWDumper
[13] The SQL dump is available on AWS S3:
https://s3-us-west-2.amazonaws.com/sqldumpsaketgreekwiki/greekwiki.sql.gz

3.2 Benchmarking Client

The Yahoo Cloud Service Benchmark(YCSB)[14] provides an open source Java based benchmarking client for evaluating the performance of a wide variety of key-value stores. It is highly efficient, capable of generating high workloads with very low CPU usage. It is also highly customizable, allowing various features be added. This was recently extended to benchmark HTTP Restful Web services[15] We use this benchmarking client for benchmarking our server. This module makes HTTP calls to the endpoints specified in the input trace files.

A trace is a sequence of URLs gathered on the basis of a metric. The selection of URLs from the trace by the benchmarking tool for querying the HTTP endpoints follows a user selected distribution. We created a read trace based on pageview statistics of the Greek Wikipedia from the same period as that of data dump. We use 10,704 of the most popular URLs based on their pageview statistics. We verified that the URL distribution is approximately Zipfian. Assuming the URL distribution of our benchmarking is consistent with that of the full month, our trace has the Zipf's constant of 0.9898, the same as that computed from the monthly statistics.

3.3 Amazon Web Services (AWS)

Amazon Web Services (AWS) cloud infrastructure is used for conducting all our experiments. AWS offers benefits of easy replication of the infrastructure and pay as you go usage. This helps in maintaining consistency across all the experiments and for replicating them.

3.3.1 Elastic Cloud Compute (EC2)

EC2 is a Web service that provides secure, re-sizable compute capacity on the cloud. We want the server CPU to be the bottleneck when the server is overloaded. We also observe that higher than 1G of network bandwidth is needed when the server is handling more than 200 requests/second. Thus we select an instance that is has high network bandwidth, high amount of memory and low CPU cores. All the parts of the Wikipedia stack except the database are thus deployed on a single EC2 instance with 8 virtual cores, 60GB of memory, and 10G of network bandwidth.

Another instance is used as a client for benchmarking purposes.

[14] https://github.com/brianfrankcooper/YCSB
[15] https://github.com/brianfrankcooper/YCSB/commit/1afb9af7ecd5db6c6927f51b3bcd3ceb21369c96

3.3.2 Relational Database Service (RDS)

We use the Amazon Relational Database Service (RDS) for hosting the database on the cloud. The resources on the database instance are ensured to be more than sufficient so that the database does not become a bottleneck.

All the EC2 and RDS instances are hosted in the same region to reduce network latency.

3.4 Automation

We use Ansible for automating our experiments. Ansible[16] is an open source automation tool that can perform a wide range of tasks like configuration management, application deployment, task automation, orchestrations etc. Various tasks in Ansible are defined using a YAML file called as a playbook.

We create an Ansible playbook[17] which uses a pre-configured AWS snapshot of the server, client and the database[18]. The snapshots have all the necessary software and tools installed. The page cache size for the server can be specified as a configuration variable for the Ansible playbook. After deploying the server, client and the database, Ansible configures all the dynamic settings for them. This includes creating the necessary security keys, updating the appropriate database endpoint on the server, updating server IP addresses on various scripts on the client, etc. For creating multiple server-client pairs with different cache sizes, the same Ansible playbook can be executed with different values of page cache.

3.5 Summary

We discussed the various design considerations involved in testing our hypothesis. Redis was a suitable candidate for implementing the archival system in the cache due to its persistence mechanism. We saw that Wikipedia proved to be ideal for performing our experiments due to its open source nature, ease of accessing its software stack, and its resemblance to real world websites. We then discussed various optimizations for the server stack in order to reduce the effect of external factors and improve performance. Further we discuss our automation techniques for deploying our test infrastructure. We look at the procedures to perform our experiments and their results in Chapter 4.

[16] https://www.ansible.com/

[17] https://github.com/VTUL/Archiving-with-InMemory-Cache.git

[18] Hosted as publicly available Amazon Machine Instances(AMI)
Server:ami-8d3fa4ed, Client:ami-1838a378, database ID: greekwiki

Chapter 4

Results

We present the results of our experiments in this chapter. Section 4.1 explains our experimentation process. We first evaluate the impact of archiving on the server for a fixed memory capacity of the page cache. Section 4.2 presents the results. We then repeat this experiment for multiple page cache sizes. Section 4.3 shows the corresponding results.

4.1 Performing the Experiments

In order to ensure that the object cache used by Mediawiki is not the bottleneck, we do not restrict its maximum memory. Before the start of experiments we query each page once without any hard timeout deadlines. We use the Linux curl tool[1] for this purpose. This process ensures that the Mediawiki object cache has the metadata for every page in the trace as the memory of the object cache is not restricted and every page is accessed at least once. This step is necessary to prevent any time consuming update operations due to cache misses in the object cache. This guarantees that the actual server performance is influenced majorly due to page cache and not the object cache. We observe that the object cache consumes a maximum memory of 1.07GB after the metadata for all the pages is stored in the cache.

Note that in this process, the page cache is updated too, but only until its maximum memory size is reached. The number of pages cached in the page cache would depend on the size of the maximum available memory configured.

Thus before the start of our experiments, we ensure that the page cache is filled to its maximum memory limit and the object cache has the metadata for all the pages in the trace.

[1] https://curl.haxx.se/docs/manpage.html

We stress test the servers by increasing the number of requests per second on every iteration. Maximum timeout for each request is set to 10 seconds. An error is defined as a timeout on a particular request or if we get a HTTP 5xx error code as response. We are interested in the maximum throughput the server permits for a particular page cache size and disk persistence configuration. The server performance with Redis persistence disabled is considered to be the base comparison in our experiments. We carry out similar experiments with Redis persistence enabled.

We use the Greekwiki dump. The following are some of the features of the dump:

- The dump, when loaded in the database, occupies 105GB of space
- There are 119440 pages having more than 475000 revisions in total. Some of the pages have more than 500 revisions.

The read trace consists of 10704 URLs. The write trace consists of 5000 URLs. Each update operation using the write trace happens every 10000 read operations. This is to ensure that the caches are regularly invalidated.

For a given page cache size, we measure the maximum server throughput and latency. As discussed in Section 3.1.2, we use the Redis "appendonly" mechanism to write the cache content to the disk. Redis is configured to archive every write operation in the page cache. Note that Redis by default overwrites the file to which the cache content is getting written if the file size gets too large. We disable this feature so as to prevent overwrites. Also, preventing duplication of archived content is left to further research.

4.2 Individual Experiments

We set up the server as described in Chapter 3. Ansible playbook can be used to deploy the EC2 instances and the database. Refer Appendix A.1.1 for exact details on how to perform the experiments. We evaluate the server performance by setting the maximum page cache memory. Section 4.1 describes the initial conditions of the server. We start by 25 operations per second and increment 25 operations per second every five minutes until we start receiving errors.

For instance, for a cache size of 300 MB, the maximum requests server can handle without errors is 75 operations per second while archiving as compared to 150 operations while not archiving. Refer Table 4.1 for more details. Figure 4.2 shows the effect on latencies due archiving.

Note that there may be random timeout errors during the course of the experiment due to the uncertainties introduced by the network. This is because the server client and the

database use a public network. The public network may be shared by other EC2 instances thus leading to network uncertainties. Disk writes (while archiving) are uncertain too and may result in errors even though the server is not at its maximum capacity. In such cases we repeat the experiment so that there are no errors till the server reaches maximum capacity.

We use general purpose Amazon Solid State Drive (SSD) for storage of archives. The performance of the server while archiving highly depends on the maximum throughput provided by the SSD. Since we are using a 250GB SSD, its default maximum throughput is 750 IOPS[2]. However, these SSD's accumulate 'credits' over time that can be used to provide a burst throughput of up to 3000 IOPS. When a new volume is created, there are enough credits to provide a 3000 IOPS throughput for 30 minutes. However, after the credits are used, the throughput falls back to 750 IOPS. We ensure that during our experiments, there are enough credits to provide 3000 IOPS. If there are not sufficient credits letting the server idle would help in increasing the number of credits, since credits accumulate over time.

4.3 Aggregated Results

We perform the experiments as discussed in Section 4.1 for different page cache sizes[3]. For every cache size the experiment is performed once and the observations are recorded. At a page cache size of 900MB, all the requests are cache hits. Thus, the cache does not get updated frequently, leading to lower disk writes.

Figure 4.1 shows the number of requests per second at which errors start showing up. With Redis persistence disabled (not archiving) we see that the server performance increases almost linearly with increase in page cache size. Higher page cache size results in larger number of pages being stored in the page cache and thus cause higher cache hits.

Higher cache hits are beneficial from the perspective of archiving since the cache would be updated less frequently. As we are writing the update operations to the disk, higher cache hits would thus imply lesser expensive disk write operations. We see from Figure 4.1 that at higher cache sizes, the performance while archiving approaches to that without archiving.

Up to a cache size of 500MB, the server maximum requests the server can handle is considerably low while archiving. See Table 4.1 for more details. However as we increase the cache size to 600MB (66% of maximum) and above, we see that the degradation in performance is not as much. The server capacity is reduced by a maximum of 50 operations/sec while archiving.

[2]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html

[3]All the raw results obtained are provided as a separate file (Results.zip) along with this document

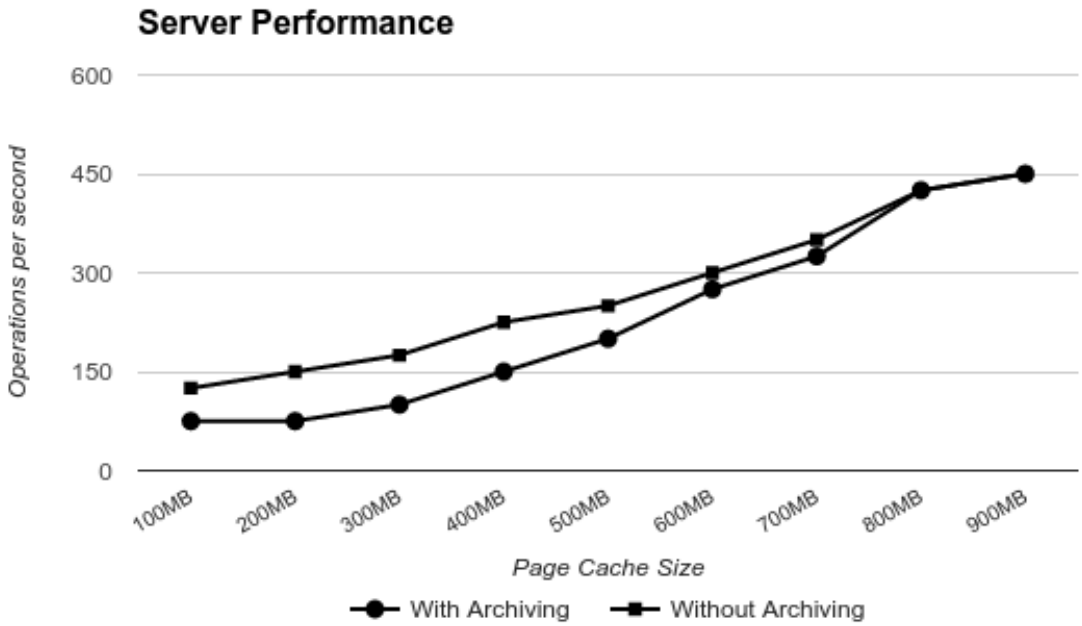


Figure 4.1: Requests/Second at which we start seeing errors

Page Cache Size(MB)	With Archiving	Without Archiving
100	50	100
200	50	125
300	75	150
400	125	200
500	175	225
600	250	275
700	300	325
800	400	400
900	425	425

Table 4.1: Maximum Requests/second the server can handle without errors

Figures 4.2 and 4.3 show the maximum latency below which 90% of the pages lie (90th percentile latencies) for each cache size while the server is operating below its limits as defined in Table 4.1. We see that the increase in latencies due to archiving is less than 30 ms. This increase is only 10 ms, for cache size greater than or equal to 600MB (66% of maximum) while the server is operating at less than 100 operations per second.

We gather various metrics for the server and the database using Amazon Cloudwatch[4]. AWS

[4]https://aws.amazon.com/cloudwatch/

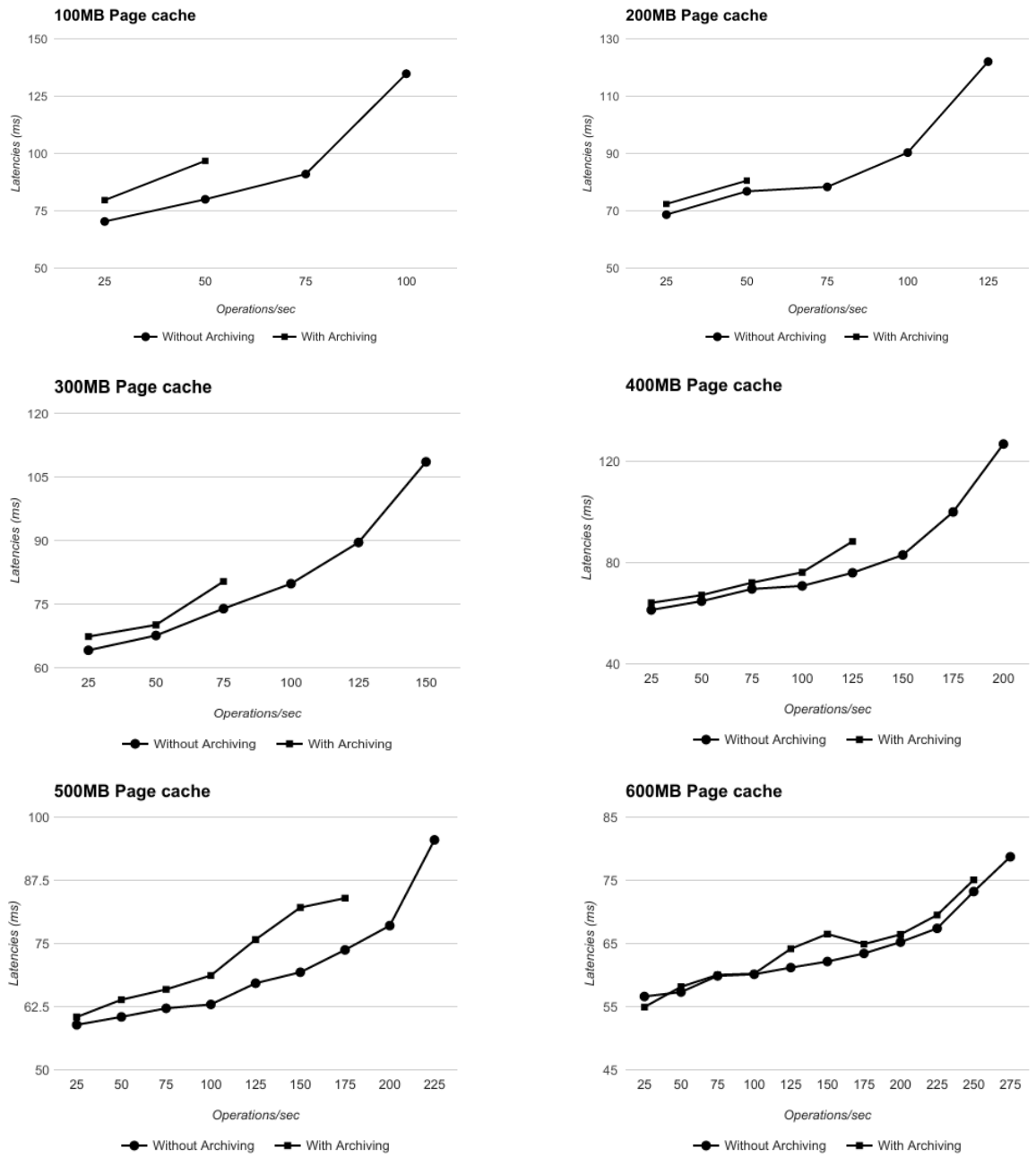


Figure 4.2: 90th Percentile Latencies for different cache sizes

provides a command line interface [5] for scripts to gather metrics in bulk. However, gathering metrics using the AWS-CLI is a very cumbersome process. Hence we use an open source Java based tool[6] for exporting the data to spreadsheet.

[5]https://aws.amazon.com/cli/
[6]https://github.com/petezybrick/awscwxls

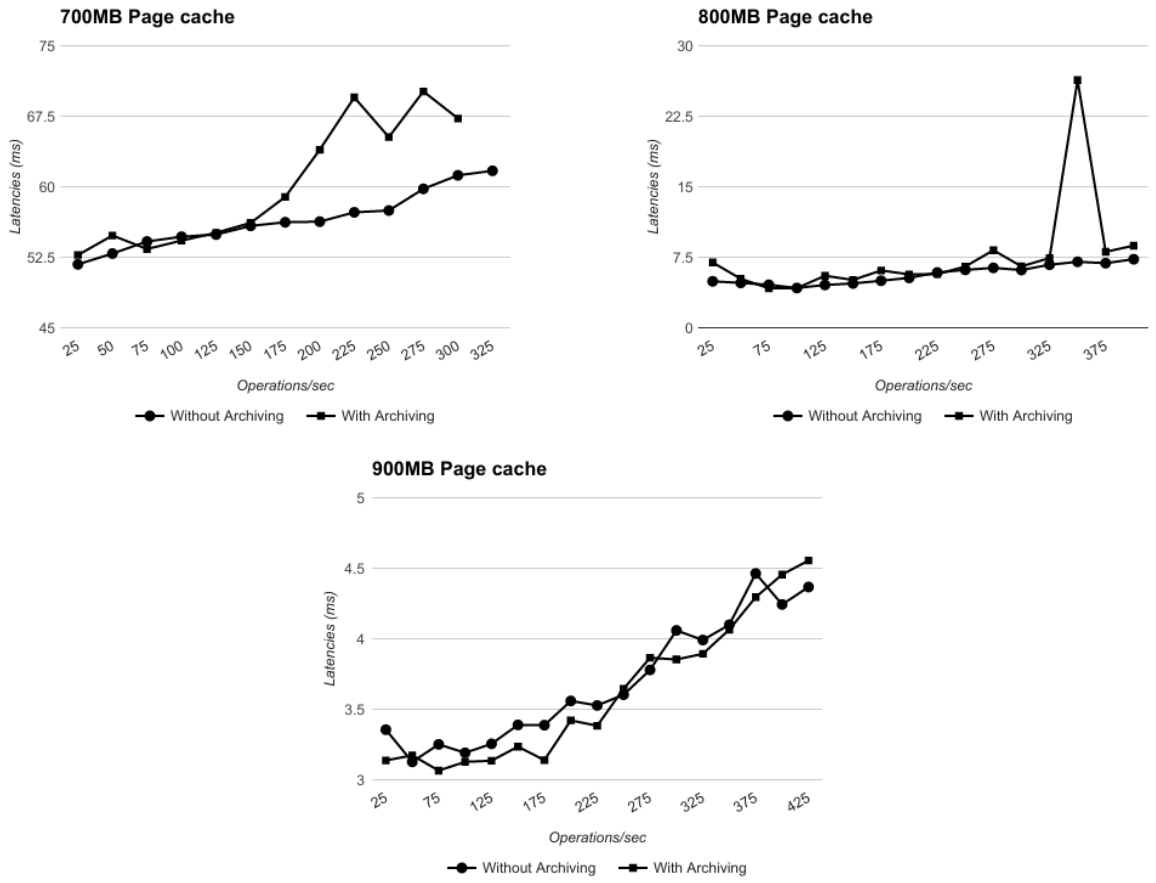
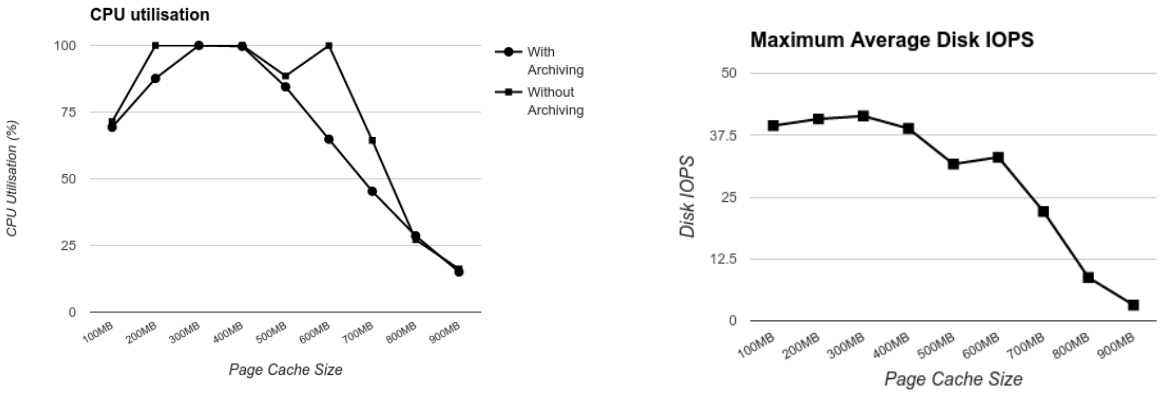


Figure 4.3: 90th Percentile Latencies for different cache sizes



(a) CPU usage

(b) Disk Usage

Figure 4.4: Performance Metrics

Figure 4.4a shows the CPU utilization of the server as a function of page cache size. The minimum resolution to get CPU utilization data is 1 minute for Cloudwatch. It may so happen that the CPU utilization peaks resulting in errors and then falls below 100%. This results in the actual value sampled by Cloudwatch to be less than 100%. We see this behaviour for cache sizes of 100MB while archiving where we observe that the CPU utilization spikes and falls below 100%.

Further, we see that as we increase the page cache size, CPU no longer remains a bottleneck. See Figure 4.4a. Rather, we start seeing errors due to several possible reasons. Network latencies cause an increase in the time needed for establishing new connections to the database. Further we use socket based communication for a lot of services like PHP, Redis object cache and Redis page cache use sockets. At higher requests, the demand for number of sockets increases thus increasing the socket latency.

Figure 4.4b shows the disk utilization metrics as measured using Cloudwatch. We use a Solid State Drive (SSD) backed volume that allows a maximum capacity of 750 IOPS for a disk of capacity 250GB that can burst up to 3000 IOPS. For SSD backed volumes, the maximum resolution for gathering the disk metrics is 5 minutes. We thus collect the number of write operations performed every 5 minutes and pick the interval with highest write operations. We use this data to compute the average IOPS shown in Figure 4.4b by dividing the number of write operations by 300 seconds. We see that the disk utilization falls as the cache size increases. Note that often a lot of write operations occur in bursts. However, since we are taking the average over a period of time, the average IOPS is low.

4.4 Summary

In this chapter, we put forward the various results we obtained by performing the proposed experiments to verify our hypothesis. We first fixed the page cache size and evaluated the maximum requests the server can handle while archiving and the effect on latency of the rendered pages. We then varied the page cache size and studied its effect on the server. The following chapter discusses the various insights we can draw from our experiments.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

To evaluate our hypothesis, we first measure the server performance without archiving in terms of its maximum latency and the number of requests per second it can handle without sending errors. We use this as a baseline comparison for evaluating our archiving technique. From Figure 4.1, we find that as we increase the page cache size, the number of requests the server can handle increases almost linearly. An increase in page cache size results in higher cache hits. This implies that lower server resources are needed as PHP processing decreases and costly database read operations are reduced.

Next, we enable Redis persistence for archiving and conduct the same set of experiments to measure server performance. We find that the maximum number of requests the server can handle is significantly lower for cache sizes less than 66% of the maximum (see Figure 4.1). We archive every update operation made to the Redis page cache. At lower cache size, higher cache misses lead to larger number of update operations thus leading to larger disk IO operations (see Figure 4.4b). Disk IO operation being expensive as discussed in Section 3.1.2 leads to performance degradation. However, for cache sizes greater than 66% of the maximum, the server performance is close to the baseline performance. From Table 4.1 we see that the maximum degradation is around 50 requests/second. This is the tradeoff to be made for archiving.

Further, our experiments show that there is a slight (on the order of a few milliseconds) increase in the latency of the rendered pages using our technique of archiving (see Figure 4.2 and 4.3). This should not have significant effect on user experience. However it should be ensured that the server load is below its maximum capacity.

5.2 Future Work

We do not filter the archived content to prevent duplication. The content can be selectively archived to improve performance further. A possible way of doing this is to extend the Redis thread responsible for persistence to archive only selective content by maintaining an internal data structure that stores key history. Since only keys are needed for finding similar content, a large amount of key history can be stored in Redis. An interesting problem to consider is to study the amount of memory needed ensuring reliable de-duplication.

It is necessary to store the content on a central data store. Using a local server storage is not scalable. Since the data is transferred over a network to the data store, the archiving performance would depend on the network bandwidth as well as the data store latency. The Redis persistence thread can be changed to write to a network data store and not the disk. The Redis Module API provides access to core internal data structures which can be used for implementing such a design.

Further, the content can be asynchronously written to the data store. This may result is loss of data if the data store buffer overflows, but should ensure higher performance at lower cache sizes, as the latency to execute archival tasks would be reduced significantly.

The Ansible playbook used to deploy infrastructure can be improved further. A virtual private cloud could be created where all the instances are deployed to reduce random errors due to network on the experiments. Also different Amazon storage volumes can be used to experiment with and study the effects of archiving.

Finally it is necessary to perform multiple iterations of these experiments and average out the results. This would help in further refining the results and remove any anomalies.

5.3 Summary

We propose a novel idea to leverage the caching infrastructure for archiving. Using the cloud infrastructure provided by Amazon Web Services (AWS) ensures that our experiments are replicable and extendable for future research. The software tools we used for deploying Wikipedia, like Web server, database, etc., are commonly used in a wide variety of Web applications. Moreover, pages are rendered in a dynamic fashion. Thus we can say that our experiments are widely applicable, increasing the validity and impact of our results.

We ensure that every component of the server stack is optimized. We try to reduce the impact of any factor that may have an external influence on our experiments. This includes avoiding access to external content for rendering pages that are not present in the database. Using Redis as a page cache provided an ideal solution for us due to its comparable performance to traditional caching tools and its persistence mechanism which we use for archiving. We also use a separate Redis instance with Mediawiki as object cache. Configuring Mediawiki to use

Redis is not straightforward due to the inbuilt locking mechanism as discussed in Chapter 3. We find that to boost performance and best use the non-blocking mechanism of Redis, the locks have to be disabled.

We show that it is possible to archive webpages using the caching infrastructure of the server. We find that the number of requests the server can handle while archiving is significantly lower for lower cache sizes as compared to not archiving. However as we increase the cache size the comparative performance improves. We also find that there is not a significant increase in the latencies of webpages as a result of archiving.

Appendix A

User Manual

A.1 Performing the Experiments

As discussed in Chapter 3, we use the AWS infrastructure to conduct our experiments. The following sections provide guidelines to perform the experiments.

A.1.1 Deploying the Infrastructure

AWS account

A valid AWS account is needed for hosting the infrastructure. Please refer AWS documentation[1] steps to create an account. On creating the account, copy the AWS credentials[2] to

```
~/.aws/credentials
```

on your local machine for remote access to AWS services.

Server

All the components except the database are hosted on a single EC2 instance. AWS offers a wide variety of instances [3] based on different CPU, memory and network requirements. Based on our design considerations described in Chapter 3, we select the r4.2xlarge instance

[1] http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/AboutAWSAccounts.html
[2] http://docs.aws.amazon.com/general/latest/gr/managing-aws-access-keys.html
[3] https://aws.amazon.com/ec2/instance-types/

offering 8 vCPU, 61GB of memory and up to 10GB of network bandwidth. We provide the fully configured amazon machine image (AMI) from which a EC2 instance can be launched.

Once the EC2 instance is up, SSH into the instance and follow the steps as described below

- Restart all services

  ```
  service mysqld stop
  service redis start
  service redisobjectcache start
  service php-fpm restart
  service nginx restart
  ```

- Update the Database endpoint

  ```
  cd /var/www/html/mediawiki/
  vi LocalSettings.php
  ```

  Change the value of $wgDBserver variable to the appropriate endpoint URL of the RDS instance.

- Access the URL (Public IP address of the EC2 instance) of the server from a browser to verify if the website is up and functioning. The Greek Wikipedia homepage should be displayed. If there issues with Nginx being not accessible try restarting PHP-FPM and NGINX again.

Cloud-watch can be enabled for accessing additional metrics. AWS-CLI[4] can be used to access the metrics. However it is difficult to extract them into a structured format. We use an open source Java application[5] for extracting the desired data into a comma separated values file.

Client

Bring up the Client similar that of the server from the AMI. A slightly lower end instance can be used for the client as the processing overhead is not high. However, ensure that there is enough network bandwidth. We use a r4.xlarge instance offering 4 vCPU, 30 GB of memory and 10G of network. Remotely access the client using SSH and execute the following steps on the server to execute the experiments.

[4]https://aws.amazon.com/cli/
[5]https://github.com/petezybrick/awscwxls

- Run scripts to fill the object cache. Update the IP address in the URL to the private IP address of the server in the bash script

```
cd /home/ubuntu/findresponsetime
vi responsetime.sh
parallel ./responsetime.sh ::: input/readtrace-[a-z][a-z]
```

- Update the necessary configuration parameters in the following files

```
/home/ubuntu/final_tests/tests/workload
/home/ubuntu/final_tests/tests/test
```

- Execute the experiments

```
./test
```

Relational Database Service (RDS)

We host the MySQL database on an Amazon RDS instance of size 500 GB. Similar to EC2, we provide an RDS snapshot from which the database can be spawned[6]. This database if fully populated with the necessary tables holding all the page information.

A.1.2 Ansible Automation

We have created Ansible scripts for creating the necessary EC2 and RDS instances. All the orchestration jobs described in section A.1.1 are executed by the Ansible scripts.

- Clone the Ansible scripts on your local machine

```
git clone https://github.com/VTUL/Archiving-with-InMemory-Cache.git
```

- Update the variable for experiments in group_vars\all

- Run the playbook

```
ansible-playbook -i hosts -i ec2.py startServer.yml
```

with parameters specified in command line

```
ansible-playbook -i hosts -i ec2.py startServer.yml \
--extra-vars "max_page_cache=104857600 max_ops=100 max_execution_time=300"
```

[6]https://aws.amazon.com/ec2/

A quick check to see if the execution was successful is to access the Web server using its public IP address from local Web browser. The deployment is successful if the Greek Wiki Homepage is displayed. Try accessing a random page for further verification. Initially the latencies would be very high (typically it takes about 10 seconds to load the homepage initially) since none of the caches have any data in them.

If for some reason, the page is unavailable try logging in remotely to the server and perform the steps described by A.1.1.

A.1.3 Running The Experiments

Running the Ansible playbook would bring up server-client-database for the configured page cache size. It would also configure them with appropriate settings and restart the necessary services. We can reach the server through our Internet browser at its assigned public IP address. After it is ensured that the server is working correctly, perform the following steps

- Fill the object cache and page cache for reasons discussed in Section 4.1

  ```
  cd /home/ubuntu/findresponsetime
  parallel ./responsetime.sh ::: input/readtrace-[a-z][a-z]
  ```

  Ensure that after the script has finished execution that the Redis object cache is filled. The execution time is about 30 minutes

  ```
  redis-cli -p 6380
  > info
  ```

  Check that the 'used_human_memory' parameter is around 1.07Gb. Next check the same parameter to verify if the page cache is filled to its maximum.

  ```
  redis-cli -p 6379
  > info
  ```

  Update the following setting in /etc/php.ini on the server

  ```
  mysql.connect_timeout = 5
  default_socket_timeout = 5
  ```

  The restart the PHP server

  ```
  service php-fpm restart
  ```

- Once the caches are filled, the experiments can be run.

```
cd /home/ubuntu/final_tests/tests/
./test
```

If for some reason, a few random errors start showing up before the desired breakdown point, stop the scripts and rerun it again. These errors are easy to distinguish because they occur usually less than 5 times during the course of experiment. When the server breaks down, we start seeing huge number of errors. We see that multiple iterations may be needed as the AWS network is an unpredictable factor in our experiments. Ensure that both caches are always filled before the start of experiments. Also ensure that there are enough burst credits while using Redis persistence to sustain 3000 IOPS for the EBS volume throughout the course of the experiment. Using Cloudwatch to monitor burst credits, CPU utilization and network may be helpful during experiments. It may also help to add delays before increasing the number of operations on the server but is not required.

A.1.4 Collecting the Results

Once experiments are finished, the results of the experiments are stored on the client machine

```
cd /home/ubuntu/final_tests/tests/output/<requests/second>
```

For example the result for 50 operations/second is stored in 'output/50' file. The results can be downloaded to local machine using some file transfer tool like 'scp'. For gathering Cloudwatch metrics, use the awscwxls. Download the properties file[7] and copy it into the 'properties' folder.

```
git clone https://github.com/petezybrick/awscwxls.git
cd run
./runcwxls properties/metrics.properties
```

Make sure the AWS account keys are updated in ~/.*aws* and the time duration to obtain the metrics is appropriately updated in the configuration file.

A.2 Mediawiki Management

All the scripts for Mediawiki maintenance are located in the 'maintenance' directory in the root Mediawiki folder. On the server

```
cd /var/www/html/mediawiki/maintenance
```

[7] https://github.com/VTUL/Archiving-with-InMemory-Cache/blob/master/Configurations/Cloudwatch

A.2.1 Importing a new Wikipedia Dump

To update the database for a fresh installation of Mediawiki, use the importDump.php script

```
php importDump.php < dumpfile.xml
```

A.2.2 Updating Indexes

To rebuild database indexes use the rebuildall.php script

```
php rebuildall.php
```

Appendix B

Developer's Manual

This appendix describes the various configurations made in addition to the default ones to optimise the server. All the configurations can be obtained from our github repository[1]

B.1 NGINX

.

The following configurations are added to optimize NGINX.

```
sendfile              on;
tcp_nopush            on;
tcp_nodelay    on;
keepalive_timeout   65;
types_hash_max_size 2048;
include             /etc/nginx/mime.types;
default_type        application/octet-stream;
client_max_body_size 20M;
```

To interact with the cache, Nginx uses the 'ngx_srcache' [2] module. This provides a transparent subrequest-based caching layout for arbitrary Nginx locations. This uses a Nginx upstream module ngx_redis2 [3] that provides a REST like interface for the srcache module. Nginx can then be used with Redis as page cache by setting the following configuration:

```
srcache_response_cache_control off;
```

[1] https://github.com/VTUL/Archiving-with-InMemory-Cache.git
[2] https://github.com/openresty/srcache-nginx-module
[3] https://github.com/openresty/redis2-nginx-module

```
srcache_store_statuses 200 302;
srcache_fetch GET /redis-fetch $key;
srcache_store PUT /redis-store key=$escaped_key;
```

Sockets are used for interacting with PHP. Use of sockets improves performance since the data transfer between PHP and Nginx occurs at the operating system level and not through network.

```
fastcgi_pass  unix:/var/run/php-fpm/php-fpm.sock;
```

B.2 Redis

B.2.1 Page Cache

In order to use Redis Persistence

```
redis-cli -p 6379
> config set appendonly yes
> config set appendfsync no
```

For disabling Redis Persistence,

```
redis-cli -p 6379
> config set appendonly no
```

Object Cache

Default Redis configuration is used for the Redis Object cache. Ensure that the maximum memory is set to a large value and persistence mechianism is disabled.

B.3 Mediawiki

- We add the following optimisations to Mediawiki:

  ```
  $wgMainCacheType = 'redis';
  $wgSessionCacheType = 'redis';
  $wgMessageCacheType = 'redis';
  ```

```
$wgParserCacheType = 'redis';
$wgCacheDirectory = "./cache/wiki";
$wgEnableSidebarCache = true;
$wgDisableCounters = true;
$wgMiserMode = true;
$wgAntiLockFlags = ALF_NO_LINK_LOCK | ALF_NO_BLOCK_LOCK;
$wgJobRunRate = 0.001;
$wgEnableTooltipsAndAccesskeys = false;
$wgSQLMode = null;
$wgUseGzip = true;
$wgHitcounterUpdateFreq = 500;
$wgTmpDirectory = "./tmp";
$wgUseInstantCommons =false;
```

- Redis is used as the object cache. We have to define the Redis object Cache within the Redis configuration files.

```
$wgObjectCaches['redis'] = array(
    'class'                 => 'RedisBagOStuff',
    'servers'               => array('127.0.0.1:6380'),
    'connectTimeout'        => 1,
    'persistent'            => true,
);
```

- To disable images we redirect image thumbnails to a 404 not found script.

```
$wgThumbnailScriptPath = "{$wgScriptPath}/thumb{$wgScriptExtension}";
$wgGenerateThumbnailOnParse = false;
```

- A redirect script has to be added to Nginx for redirecting these images.

```
location /images {
        location ~ ^/images/thumb/(archive/)?[0-9a-f]/[0-9a-f][0-9a-f]/
```

```
                        ([^/]+)/([0-9]+)px-.*$ {
        rewrite ^/images/thumb/[0-9a-f]/[0-9a-f][0-9a-f]/([^/]+)/([0-9]+)px-.*$
                        /thumb.php?f=$1&width=$2;
        rewrite ^/images/thumb/archive/[0-9a-f]/[0-9a-f][0-9a-f]/([^/]+)
            /([0-9]+)px-.*$ /thumb.php?f=$1&width=$2&archived=1;
    }
    include /etc/nginx/fastcgi_params;
    fastcgi_param SCRIPT_FILENAME    $document_root/thumb.php;
    fastcgi_pass  unix:/var/run/php5-fpm/www.sock;
}
```

B.4 Database

Default RDS MySQL configuration is used for database.

B.5 Operating System

The following optimizations[4] are made to the server for handling higher loads.

- Increase file handle limit

  ```
  sysctl -w fs.file-max=100000
  ```

- Increase the number of backlogged sockets

  ```
  net.core.somaxconn = 1024
  ```

- There's a kernel parameter that determines how long a migrated process has to be running before the kernel will consider migrating it again to another core. It is reduced from 5 microseconds to 5 nanoseconds.

  ```
  sysctl -w kernel.sched_migration_cost_ns=5000000
  ```

[4] https://tweaked.io/guide/kernel/

Bibliography

Allan Arvidson and Frans Lettenstrm. 1998. The Kulturarw Project The Swedish Royal Web Archive. *The Electronic Library* 16, 2 (1998), 105–108.

Steve Bailey and Dave Thompson. 2006. Building the UK's first public web archive. *D-Lib Magazine* 12, 1 (2006), 1082–9873. `http://www.dlib.org/dlib/january06/thompson/01thompson.html`

Yacine Boufkhad and Laurent Viennot. 2003. *The observable web*. Ph.D. Dissertation. INRIA.

Brian E Brewington and George Cybenko. 2000. Keeping up with the changing web. In *Computer*, Vol. 33. IEEE, 52–58.

Justin F. Brunelle, Michael L. Nelson, Lyudmila Balakireva, Robert Sanderson, and Herbert Van de Sompel. 2013. Evaluating the SiteStory Transactional Web Archive with the ApacheBench Tool. In *Research and Advanced Technology for Digital Libraries: International Conference on Theory and Practice of Digital Libraries, TPDL 2013*. Springer, 204–215. `DOI:http://dx.doi.org/10.1007/978-3-642-40501-3_20`

Junghoo Cho and Hector Garcia-Molina. 1999. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*. `http://ilpubs.stanford.edu:8090/376/`

Herbert Van de Sompel, Michael L. Nelson, Robert Sanderson, Lyudmila Balakireva, Scott Ainsworth, and Harihar Shankar. 2009. Memento: Time Travel for the Web. *CoRR* abs/0911.1112 (2009). `http://arxiv.org/abs/0911.1112`

Dimitar Denev, Arturas Mazeika, Marc Spaniol, and Gerhard Weikum. 2011. The SHARC Framework for Data Quality in Web Archiving. *The VLDB Journal* 20, 2 (April 2011), 183–207. `DOI:http://dx.doi.org/10.1007/s00778-011-0219-9`

Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. 2003. A Large-scale Study of the Evolution of Web Pages. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*. ACM, New York, NY, USA, 669–678. `DOI:http://dx.doi.org/10.1145/775152.775246`

Kent Fitch. 2009. Web site archiving - an approach to recording every materially different response produced by a website. (2009). http://www.nla.gov.au/openpublish/index.php/nlasp/article/view/1258

Daniel Gomes, João Miranda, and Miguel Costa. 2011. A Survey on Web Archiving Initiatives. In *Research and Advanced Technology for Digital Libraries: International Conference on Theory and Practice of Digital Libraries, TPDL 2011*. Springer, Berlin Heidelberg, 408–420. DOI:http://dx.doi.org/10.1007/978-3-642-24469-8_41

Allan Heydon and Marc Najork. 1999. Mercator: A Scalable, Extensible Web Crawler. *World Wide Web* 2, 4 (April 1999), 219–229. DOI:http://dx.doi.org/10.1023/A:1019213109274

Internet Systems Consortium ISC. 2016. ISC Internet Domain Survey. (2016). Retrieved April 16, 2017 from https://www.isc.org/network/survey/

International Telecommunication Union ITU. 2016. ICT Facts and Figures. (2016). Retrieved April 16, 2017 from http://www.itu.int/en/ITU-D/Statistics/Pages/facts/default.aspx

Wallace Koehler. 2004. A longitudinal study of Web pages continued: a consideration of document persistence. *Information Research* 9, 2 (2004). http://www.informationr.net/ir/9-2/paper174.html

Julien Masanés. 2006. *Web Archiving: Issues and Methods*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–53. DOI:http://dx.doi.org/10.1007/978-3-540-46332-0_1

Gordon Mohr, Michael Stack, Igor Ranitovic, Dan Avery, and Michele Kimpton. 2004. An Introduction to Heritrix An open source archival quality web crawler. In *4th International Web Archiving Workshop*. Springer Press.

Netcraft. 2015. Web Server Survey. (2015). Retrieved 23 April 2017 from https://news.netcraft.com/archives/2015/09/16/september-2015-web-server-survey.html

Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. 2004. What's New on the Web?: The Evolution of the Web from a Search Engine Perspective. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, 1–12. DOI:http://dx.doi.org/10.1145/988672.988674

Christopher Olston and Sandeep Pandey. 2008. Recrawl Scheduling Based on Information Longevity. In *Proceedings of the 17th International Conference on World Wide Web (WWW '08)*. ACM, New York, NY, USA, 437–446. DOI:http://dx.doi.org/10.1145/1367497.1367557

Margaret Phillips. 2009. PANDORA, Australia's Web Archive, and the digital archiving system that supports it. *National Library of Australia Staff Papers* (2009).

V. Shkapenyuk and T. Suel. 2002. Design and implementation of a high-performance distributed Web crawler. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 357–368.

Marc Spaniol, Dimitar Denev, Arturas Mazeika, Gerhard Weikum, and Pierre Senellart. 2009. Data Quality in Web Archiving. In *Proceedings of the 3rd Workshop on Information Credibility on the Web (WICOW '09)*. ACM, 19–26. DOI:http://dx.doi.org/10.1145/1526993.1526999

Diomidis Spinellis. 2003. The Decay and Failures of Web References. *Commun. ACM* 46, 1 (Jan. 2003), 71–77. DOI:http://dx.doi.org/10.1145/602421.602422

H. Van de Sompel. 2012. *SiteStory transactional web archive software released*. D-Lib Magazine 18.9/10. http://www.dlib.org/dlib/september12/09inbrief.html