UNIVERSITETET I AGDER

A novel spatio-temporal scheme for reducing the rate of false positives in Bloom Filter based URL-caching

By

Thoams Hansen Gøytil, Andreas Lundberg

Thesis submitted in Partial Fulfillment of the Requirements for the Degree Master of Science in Information and Communication Technology

Faculty of Engineering and Science
University of Agder

Grimstad,
May 2010

Abstract:

Achieving efficient use of available resources is an important problem in the field of web mining. Monitoring and analyzing the web is extremely resource demanding, and therefore, more efficient use of resources often translates directly into improved web monitoring coverage and accuracy. One important sub problem is to reduce the memory consumption of the URL cache in a web crawler system. Utilizing the space efficient data structure Bloom filter as URL cache, will reduce the memory consumption. However, the Bloom filter introduces false positives, leading to loss of valuable web content when the filter are utilized as a URL cache in a web crawler system. Based on the latter problems of false positives, this thesis propose three novel strategies, namely a temporal, a spatial and a spatio-temporal strategy, each aiming to reduce the false positive rate introduced by the Bloom filter. During testing and evaluation of the strategies, we discovered both the spatial and temporal strategy is able to reduce the false positive in the Bloom filter. The two former strategies was then combined to test if it is possible to further decrease the false positive probability. Testing and evaluation of the combined strategies shows that it does yield a reduction in the false positive probability.

Preface

This master thesis is submitted in partial fulfillment of the requirements for the degree of Master of Technology in Information and Communication Technology at the University of Agder, Faculty of Engineering and Science. Supervisor on this master thesis have been associate professor Ole-Christoffer Granmo.

We would like to give a thank to Ole-Christoffer Granmo for excellent advice and guidance throughout the whole period of the project. His feedback throughout the project have been of great value. We would also like to thank our co-supervisor Jaran Nilsen at Integrasco A/S for guidance and help.

Grimstad, May 2010.

Thomas Hansen Gøytil and Andreas Lundberg

Thesis definition

A novel spatio-temporal scheme for reducing the rate of false positives in Bloom Filter based URL-caching

Group 05: Thomas H. Gøytil and Andreas Lundberg

Achieving efficient use of available resources is an important problem in the field of web mining. Monitoring and analyzing the web is extremely resource demanding, and therefore, more efficient use of resources often translates directly into improved web monitoring coverage and accuracy. One important sub problem is to reduce the memory consumption of the URL cache. A URL cache, in this context, is utilized by a web crawler system to keep track of all visited URLs to prevent the web crawler from unnecessarily re-visiting web pages. The URL cache may contain large amount of URLs in a typical web crawling session, making it a bottleneck performance wise.

The purpose of this thesis is to propose a novel solution to the latter problem. The solution involves a space efficient data-structure called the Bloom filter. In brief, the Bloom filter is used as an URL cache with the purpose of reducing memory consumption. The Bloom filter is based on hash functions, which means that it is prone to produce false positives. In this context, this, in turn, means that the Bloom filter may falsely state that an URL can be found in the URL cache when it is not actually there. Such mistakes may block the crawler from subsets of the web. Accordingly, a further purpose of this master thesis is to investigate methods that reduce the rate of false positives in the Bloom filter when used as an URL cache in a web crawler system.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The first section briefly presents web crawler systems and the URL cache of a web crawler. Further, we present the space efficient data-structure Bloom filter and look at it in the context of a web crawler environment. In this context, the Bloom filter is used as an URL cache. We end the first part of the introduction chapter by studying benefits the Bloom filter introduces, as a motivation.

In the next section we review related work, see section 1.2. The related work section consists of the most significant published papers related to the topic and problem of this master thesis, among other, “Space/Time Trade-offs in Hash Coding with Allowable Errors”[6]. [6] was published by Burton H. Bloom in 1970. In this publication, Bloom introduces a space efficient data-structure with trade-offs in both time and space, later to be known as the Bloom filter.

The problem statement is presented in section 1.3. The section also discuss the problem in depth to clarify the main aspects of the problem.

In section 1.4, the contribution of our different approaches into the field of data-mining is evaluated. The most important topic discussed in this section is the importance of reducing the false positive rate for a Bloom filter used in the context of a web crawler to limit the loss of valuable web page content.

Before we end the introduction chapter by pinpointing the target audience and presents the report outline, the research questions are presented. The research questions are explained thoroughly, and a sub-question is also presented.

1.1 Background and Motivation

One of the oldest problems in the field of computer science is the lack of hardware resources. Hardware resources can increase cost and complexity of a system, thus it is not always feasible to just add more hardware resources. This problem still applies, although the availability of hardware resources continue to increase at a high rate.

One of the areas concerning this problem is the capacity of dynamic storage space. Dynamic storage space need some reserved hardware resources, that does not exceed the total available hardware resources. An example of such dynamic storage is a cache storage in a running application, that is used to keep track of a large amount of data.

1.1.1 Cache systems

A cache is a collection for storing temporary data to enable rapid access. To enable rapid access to the data, the cache system store the data in memory. The amount of data are limited to the dynamic storage space available to the application. A cache system is utilized by applications where data is expensive to retrieve, such as high computational cost, high access time or slow bandwidth. Applications that need to keep track of their history also utilizes a cache.

Caching is applied in many fields of the computer world. Some examples are: Hardware caches like the internal Central Processing Unit (CPU) cache, networking caches like the Domain Name System (DNS) server which maps domain names to Internet Protocol (IP) addresses and memorization caches like an URL cache in a web crawler system.

As mentioned above, the importance of caching is to provide rapid access to data, which have a high probability of being requested more than once. Caching is performed to reduce access time and reduce expensive processing. Since both access time and processing cost can be reduced by applying caching, caching can be defined as an optimization technique.

1.1.2 URL cache in a web crawler system

This thesis deals with how elements can be cached in a space efficient manner. One way to achieve this is to apply a space efficient data structure. An example of a field of research such a data structure could be applied to, is the field of web

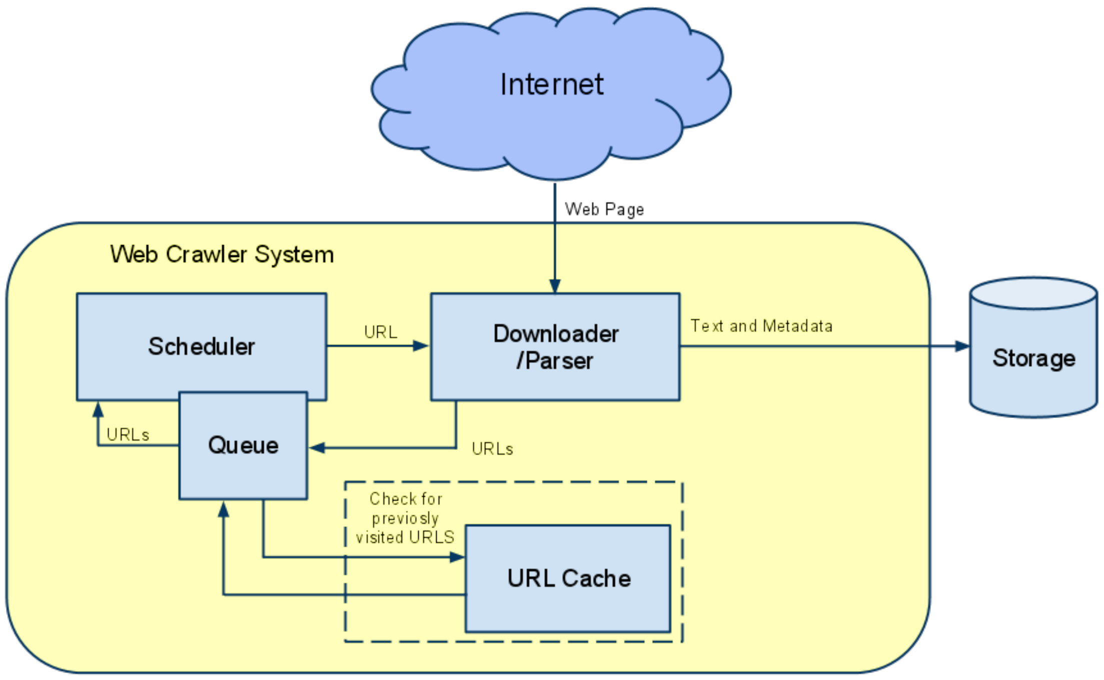

Figure 1.1: The different components of a web crawler system and how they interact.

crawler technology.

Web crawlers is used to download web pages and documents from the Internet to extract information.

A web crawler system, as shown in figure 1.1, need to keep track of all visited web pages. Keeping track of visited web pages is necessary to prevent the web crawler from unnecessarily visiting already visited web pages. As a worst case scenario, without any technique to keep track of the URLs linking to already visited web pages, the web crawler can go into a loop between only a few web pages.

Further, this loop can result in that the web crawler only visiting the same pages over and over again. Hence, if such a scenario occurs, it will, arguably, result in the web crawler only covering a small part of the web pages it was supposed to visit.

Another issue with not keeping track of visited URLs, is that the web crawler have no indication on when, or even if, all the web pages supposed to be visited are covered. In most cases, this will lead to that the web crawler never finishes.

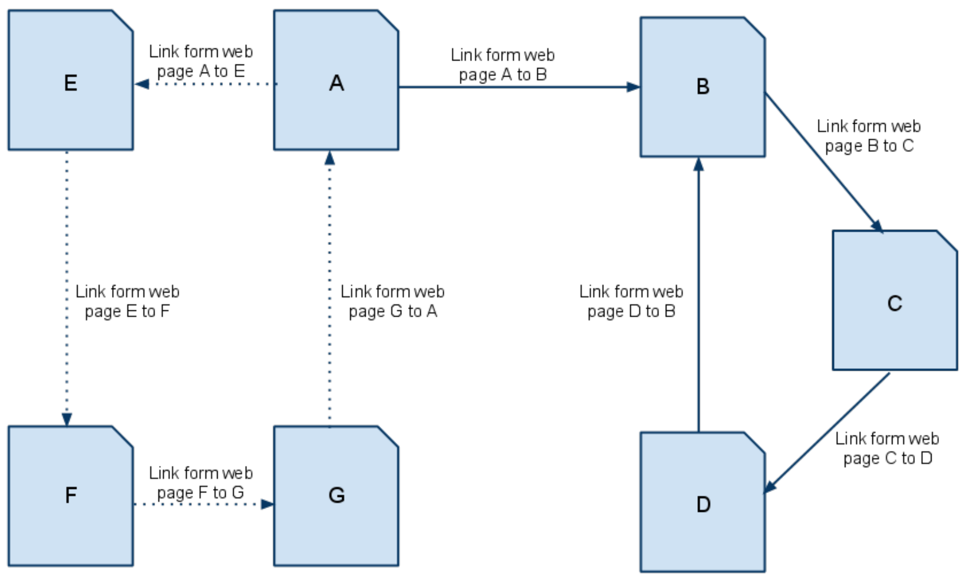
E
Link form web page A to E
A
Link form web page A to B
B
Link form web page B to C
C
Link form web page E to F
Link form web page G to A
Link form web page D to B
Link form web page C to D
F
Link form web page F to G
G
D

Figure 1.2: This figure illustrates the behavior of a web crawler system with no URL cache. The dotted lines illustrate links never visited by the web crawler.

Figure 1.2 illustrates the behavior of a web crawler system with no way of keeping track of visited web pages. The web crawler is set to visit web page A. A contains two URLs, linking to web page B and E. The web crawler then visit B, which only contain one URL linking to C. As the figure 1.2 illustrates, the crawler will then visit page C, then page D and then B.

The issue first occurs when web page B is visited for the second time. In most cases, it is not desirable for a web crawler to visit the same web page more than once in the same web crawling session. Further, web crawler will continue visiting web page C, D and B all over again in a never ending loop. Hence, web page E, F and G, also illustrated in figure 1.2, will never be visited.

Therefore, to prevent the web crawler from “looping”, a URL cache could be applied. The URL cache is thoroughly explained in section 2.6 in the web crawler chapter. Figure 1.1 shows an example of a web crawler system with such a URL cache. Each URL that the web crawler visits will be stored in the URL cache.

The URL cache needs to be able to store a large amount of URLs. This is due to the large amount of URLs the web crawler system needs to handle. From a memory consumption point of view, to store such a large amount of URLs demands a lot of available memory for the web crawler system. To reduce the memory usage of the URL cache we can utilize a space efficient data-structure.

1.1.3 Bloom filter: A space efficient data-structure

The Bloom filter, which is thoroughly investigated in chapter 3, is a space efficient randomized data-structure that supports membership queries [11]. The fact that the Bloom filter is a space efficient data-structure, implies, that it consumes less memory than a regular data-structure. Hence, the purpose of using the Bloom filter is to reduce the memory consumption.

The main components of a basic Bloom filter is a bit-array, see section 3.2, and k number of hash functions, see section 3.3. When adding an element to the Bloom filter, each of the hash functions generates a index from the given element and sets the bit at the given index position to 1 in the bit-array.

It is important that all of the k number of hash functions utilized the Bloom filter are different from each other. If two or more of the k number of hash functions used by the Bloom filter are the same, they will produce the same output. Hence, they will set the bit of the same index to 1 when adding a given input element to the Bloom filter. In other words, if two or more of the k number of hash function are the same, one or more hash functions would be redundant.

The Bloom filter may appear as a near perfect data-structure to utilize as a cache, but there is a drawback. Hash functions are prone to give collisions. Collisions, which are explained in section 3.3.2, occur when a hash functions generate the same value for two different input elements. The fact that Bloom filter is based on hash functions, introduces the issue of false positives.

In the context of Bloom filters, false positives means that the filter may falsely state that an element can be found in the filter when it is not actually there. When applied as a URL cache in a web crawler system, a false positive can lead to loss of valuable content for the web crawler.

To provide a better understanding of the problem, let us consider the following scenario:

A web crawler system uses a Bloom filter based URL cache to keep track of the visit history of the crawler. The web crawler asks the URL cache if a given URL has been visited. A false positive occur and it tells the crawler that the URL have already been visited.

The false positive will then result in the web crawler system never visiting the given URL, hence potential valuable content, on the web page the given URL links to, is never retrieved. False positives are thoroughly explained in section 3.5.2.

1.1.4 Motivation

Achieving efficient use of available resources is an important problem in the field of web mining. Monitoring and analyzing the web is extremely resource demanding, and therefore, more efficient use of resources often translates directly into improved web monitoring coverage and accuracy.

Memory resource optimization is a prevailing part of this problem. One important sub problem is to reduce the memory consumption of the URL cache. A URL cache, in this context, is utilized by a web crawler system to keep track of all visited URLs to prevent the web crawler from unnecessarily re-visiting web pages. The URL cache may contain a large amount of URLs in a typical web crawling session, making it a bottleneck performance wise.

Integrasco A/S is a company that keeps track of a vast amount of data published on the Internet. The company act as a specialized centralization point that keeps track of a subset of the resources found on the Internet. Integrasco A/S is retrieving several millions unique user written entries each month. To keep all this data consistent, web crawler systems are used to crawl web pages for new information periodically. This demands a lot of resources regarding network bandwidth, processing capacity and memory usage.

The Internet Archive web crawler, described in [7], utilizes a Bloom Filter to check if a URL have been visited before or not. The probability of a false positive will be very small in the beginning, since the number of visited URLs is very small. When the number of visited pages increase, the number of URLs increases, and the false positive probability will also increase. This solution can be practical for a small set of URLs, but for a larger set the false positive probability may grow beyond what is feasible. [7].

Finally, memory usage is one of the biggest limitations for a web crawler system, specially for the URL cache, as mentioned above. In a web crawling system, which needs to handle a large amount of web pages, the memory available is the crucial factor that decides if the web crawler is able to visit all web pages it is supposed to visit.

1.2 Related work

This section presents the most significant published papers related to the topic and problem of this master thesis. Each paper is presented with a citation from which journal or article the paper can be found. General weaknesses of the solutions are also presented.

- In the paper "Space/Time Trade-offs in Hash Coding with Allowable Errors" [6], Burton H. Bloom introduces a space efficient data structure with trade-offs in both time and space. The big disadvantage compared to a regular data structure is that it allow errors, also known as false positives. The latter space efficient data structure was later given the name Bloom filter.

  Since the Bloom filter have a certain probability of returning a false positives, it does not fit in all context. One example is an URL cache in a web crawler system, that need to handle a large amount of URLs. As the number of URLs increase, the false positive probability will also increase. This will result in that the filter will have a high probability of returning a false positive, resulting in that some web pages is never visited.

- Another significant paper is "Network applications of bloom filter: A survey" [8]. This survey gives an overview of where the Bloom filter have been used and modified in different network problems. It also provides a unified mathematical framework for understanding the Bloom filter.

- The paper "Optimizing Data Popularity Conscious Bloom Filters"[25] study the problem of minimizing the false positive probability of a Bloom filter by adapting the number of hash functions used on each data object based on the popularity of the given data object.

  As a result, [25] propose a popularity conscious Bloom filter. This Bloom filter reduces the false-positive probability, or reduces the Bloom filter memory consumption (if the same false positive probability threshold, as a basic Bloom filter, is satisfied).

  The reduction of the false positive probability, introduce some new major drawbacks. One drawback is the additional offline computational cost. The additional computational cost is introduced by a object importance metric which calculates the popularity of each data object and stored in a generated hash scheme.

  Another drawback the overhead the hash scheme introduces to the popularity conscious Bloom filter.

The last major drawback is that the object importance metric generates the hash scheme offline. That it is offline means that it is processed in a separated operation. This separated operation needs to be processed before the popularity conscious Bloom filter can be used, and produces a static hash scheme.

Due to that Internet pages and domains are highly dynamic when it comes to popularity, the popularity of a domain can in many cases increase or decrease rapidly. This gives a static representation of a the popularity a big disadvantage. Hence, the object importance metric mentioned above, needs to regenerate the hash scheme fairly often to ensure the hash scheme are up to date.

Therefore, the strategy of the Bloom filter is based on the object importance metric hash scheme, meaning that a change in the hash scheme would changes the behavior of the Bloom filter. The Bloom filter would no longer be reliable if the hash scheme is regenerated to update the popularity. This can be explained by, if the popularity for an object increase or decrease enough, the number of hash functions used to query the Bloom filter would change. This leads to that the references stored in the Bloom filter become useless.

Although the popularity conscious Bloom filter reduces the false positive rate and provide a space efficient solution, the drawbacks described above are not desirable. Especially the drawback concerning the offline computation of the hash scheme, that generates a static hash scheme. Hence, to use the popularity conscious Bloom filter in a web crawler context would demand a great amount of computational resources or a hash scheme which is not up to date.

1.3 Problem statement

Elaborated from the issues of false positives and memory consumption described in section 1.1, the problem is stated as follows:

"Use the space efficient data-structure Bloom filter to minimize the memory consumption of the URL cache in a web crawler system and propose novel spatio-temporal strategies to reduce the false positive rate introduced by the Bloom filter."

Based on the problem statement, we will investigate how to utilize a Bloom filter as an URL cache, in the context of a web crawler system. We will then, based on the investigation, propose a spatial and a temporal strategy together with an spatio-temporal strategy for reducing the false positives introduced by the Bloom filter.

To clarify the major aspects of the problem, two sub problems is derived from the problem statement and explained below.

1. **How to limit the loss of web pages, introduced by the false positive rate in the Bloom filter, in a web crawler system**

   The Bloom filter, which is briefly explained in section 1.1.3, is prone to produce false positives. Each false positive produced by the Bloom filter, when used as an URL cache in a web crawler system, will result in that a web page is never visited by the web crawler. Since the web page is not visited, the content of the given web pages is never retrieved, and are therefore regarded as lost. Therefore, we will investigate the problem of reducing false positives with the intention of limiting the loss of web page content.

   Based on the problem of false positives, we will propose novel spatio-temporal strategies to reduce the loss of content. Our strategies will be implemented and tested in a simulated web crawler environment.

2. **How to minimize the memory consumption for an URL cache, compared to a regular data structure**

   An URL cache in the context of a web crawler system needs to handle a large amount of URLs due to the large amount of URLs a given area of the web can contain. As mentioned in section 1.1.4, memory usage is one of the biggest limitations for a web crawler system. Therefore, it is of great importance to find solutions to reduce memory consumptions of the URL cache, such that web crawlers are able to handle such large amount of URLs.

   One solution to the latter problem is to utilize a space efficient data structure to minimize the memory consumption for the URL cache. The Bloom filter is such a data structure. Therefore, the space efficient data structure Bloom filter will be used.

1.4 Contributions

This section describes the contribution of our different approaches into the field of data-mining.

In this thesis, we investigate strategies to reduce the false positive rate produced by the space efficient data structure Bloom filter when utilized as an URL cache in a web crawler system. Different novel strategies are proposed, each with the intention of reducing the false positive rate compared to an URL cache utilizing a single Bloom filter.

1.4.1 A temporal strategy for reducing false positives

We propose a temporal strategy, for web crawler systems, that uses a single Bloom filter as a URL cache to keep track of URLs of previously visited web pages. The purpose of the temporal strategy is to minimize the loss of visited content missed by the web crawler, when utilizing a Bloom filter as URL cache, over a period of time.

The importance of such a temporal strategy can be justified by the continues growth of the web [5]. It is important for companies working with web crawling and search engine technology to obtain scalable solutions to cope with the growth of the web. Companies, within this field of technology, continuously crawls a set of web pages where such a strategy applies.

1.4.2 A spatial strategy for reducing false positives

We also propose a spatial strategy, which is a combination of multiple Bloom filters. The combination of multiple Bloom filters will take advantage of the fact that a Bloom filter never returns a false negative [11]. The idea is to query all the Bloom filters and compare the answers, and only give a positive answer if all filters give a positive feedback.

The intention of the spatial strategy is to investigate if it is possible to limit the false positive rate compared to a single Boom filter solution by assembling several Bloom filters together as a multiple Bloom filter solution.

1.4.3 A spatio-temporal strategy for reducing false positives

The last strategy proposed is the spatio-temporal strategy. The spatio-temporal strategy is a combination of the spatial and the temporal strategy. The intention of this strategy is to take advantage of the reduction of the false positive rate, if any, both the spatial and temporal strategies provide. Hence, if both solutions leads to a reduction of false positive, we presume that a combination of both strategies will lead to a further reduction of false positives.

1.4.4 Empirical analysis of the Bloom filter

Through empirical evaluation of the memory usage of a single and multiple Bloom filter(s), we will measure memory consumption. The memory consumption of the Bloom filter(s) will be compared to the memory consumption of regular data-structures. These regular data-structures, will, of course, be suitable to utilize as a URL cache.

1.4.5 Novel application of Bloom Filter and crawling

Our contribution to the field of data mining will be to apply the different strategies, proposed in this thesis, to create a Bloom filter solution utilized as an URL cache in a web crawler system. The latter strategies will, presumably, reduce the false positive rate introduced by the Bloom filter. Hence, reducing the false positive rate for a Bloom filter solution utilized as an URL cache in a web crawler system, will lead to reduction of loss of valuable web content for the web crawler.

1.5 Research questions

To clarify the area of focus in this thesis, we have elaborated the following research questions. These research questions are derived from the proposed strategies in section 1.4.

- **Is it possible to reduce the false positive rate by applying a novel temporal strategy which utilize a Bloom filter, containing a different combination of hash functions, upon a re-visit:**

Since an occurrence of a collision in a hash function, utilized in a Bloom filter, presumably, can lead to a false positive in the Bloom filter, it is important to consider different hash functions. Therefore, we will investigate if it is possible to apply a temporal strategy to reduce the false positive rate of a single Bloom filter.

The temporal strategy, briefly explained in section 1.4.1, uses a re-visiting strategy approach. The intention of this approach, is to reduce the false positive rate over time, by changing the combination of hash function utilized by the Bloom filter upon re-visit.

- **Is it possible to reduce the false positive rate by applying a novel spatial strategy, using multiple Bloom Filters, each with a different combination of hash functions:**

  The Bloom filter will never return a false negative when queried. In other words, a Bloom filter returning false when queried, returns a correct answer. This means that it is certain that the queried element is not present in the filter. We will therefore, by taking advantage of the previous stated fact, investigate if it is possible to reduce the false positive rate by applying the proposed novel spatial strategy. The proposed spatial strategy is briefly explained in section 1.4.2.

  The spatial strategy will utilize a combination of multiple Bloom filters. When querying if a URL is present in the Bloom filter, all the filters will be queried. If all the Bloom filters returns true, the queried URL should be considered as being present in the filter. If the filters returns an ambiguous answer, meaning that one or more of the filters returns that the queried URL is present in the filter, and one or more returns that the queried URL is not present in the filter, an occurrence of false positive is discovered.

  Since a Bloom filter never returns a false negative, as explained above, each of the latter filters that returns true, returns, presumably, a false positive. Hence, the queried URL are assumed not to be present in the Bloom filter.

  To be able to determine if an reduction of the false positive rate is obtained, we will compare the results against a one filter solution.

- **Is it possible to decrease the false positive rate further by combining the proposed spatial and temporal strategies than each of the strategies achieve on their own:**

  Each of the proposed strategies, spatial and temporal will, presumably, reduce the false positive rate compared to a single Bloom filter solution.

Therefore, we will investigate the possibility of decreasing the false positive rate further by combining the two strategies. This strategy is briefly explained as a spatio-temporal strategy in section 1.4.3.

As mentioned in section 1.1.3, the Bloom filters contains k number of hash functions. There is an adage telling that a solution is never better than the weakest link. With the latter adage in mind, we can assume that, in the first place, is better to use the best suitable hash function k times, than k different hash functions.

Replacing the k number of hash functions with the best suitable hash function, alone, would not be a good idea. The obvious reason is that, by using the best suitable hash function k times in a Bloom filter would give the same result as using it one time, as it gives the same result each time.

To take advantage of the best suitable hash function, different salts can be used to seed URLs before they are hashed. A salt can, for instance, be a string or a numeric value.

To seed the URL with a salt can be performed simply by applying a function to concatenate the salt and the URL. The purpose of seeding the URL is to get a different output for each salt, using k number of salts instead of k number of hash functions. Therefore a sub-research question is presented, concerning all of the research questions above:

- **Will a Bloom filter utilizing the same hash function a given number of times, seeded with different salts each time, give close to the same false positive rate as a Bloom filter utilizing the same given number of different hash functions:**

  Using different salts to seed the URLs will, arguably, give different output for each salt. We will therefore investigate if using different salts as seed in one hash function k times, can be as good as using k different hash functions. The motivation for this question is to be able to use the single most suitable hash function for the problem, instead of the k most suitable hash functions.

1.6 Target audience

The target audience of this thesis is anyone that is interested in resource optimization and data structures. Especially anyone who are interested in memory optimization in the context of data-mining. For the reader to fully understand the content of this thesis, a good understanding of common basic elements within the field of computer science is recommended.

1.7 Report outline

The outline of the rest of this this is elaborated with the intention of providing the reader with a informative and interesting reading experience.

Chapter 2 provides the reader with the knowledge needed to grasp the concepts of web crawler systems and web crawling strategies. The chapter provide a brief introduction to the architectural overview of a web crawler system in section 2.2, and explains the purpose of each element. The URL cache are explained in detail in section 2.6, due to its importance to the problem.

Chapter 3 introduces the Bloom filter and gives a thorough explanation of the components that the Bloom filter consist of. Hash functions is also discussed in detail in section 3.3.

Chapter 4 describes the two proposed strategies, namely the spatial, the temporal. A combination of these former strategies, a spatio-temporal strategy, is also proposed. Each strategy is described in detail.

Chapter 5 explains the test configuration and the different types of tests that was conducted. Also the results for each strategy is presented along with.

Chapter 6 discusses the results and observations of the test results.

Chapter 7 concludes the thesis and summarizes the work. Some suggestions for further work is also proposed.

Chapter 2

Web Crawler systems

This chapter starts with an introduction to web crawler systems in section 2.1. The web crawler system is an automated computer program which is used to browse and monitor the web for information.

There exists numerous architectures for web crawler systems. In the paper [9] a typical high-level architecture is presented. Figure 1.1, in chapter 1, shows an example of a typical high-level architecture similar to the one found in [9].

Downloader/parser, scheduler, queue, storage and cache are the main components in a web crawler architecture. The latter components are briefly explained in the architectural overview section, see section 2.2.

Different web crawler systems have different purposes. Some web crawler systems are used solely for discovering new web pages on the web, while others are used to collect e-mails for the more dodgy business of spamming. A web crawler strategy is utilized to fulfill the latter purpose. In section 2.3, different web crawling strategies are explained.

To enable a user good full-text search functionality on a collection of downloaded web documents, based on their content and meta data, some kind of processing of the data must be performed. The most common way of processing the data is by applying a indexing technique. Indexing is an optimization technique which enables fast search on big collections of data. Indexing is investigated in section 2.4.

Internet services such as search engines, notification, maintenance and analytic services, are all services which utilize a vast amount of information located all over the web. To be able to cope such vast amounts of information, a web crawler system is used. Section 2.5 takes a closer look on different applications of the web crawler system.

Since a web crawler system must handle a large amount of URLs, that is accessed each time the crawler visits a new web page, it is convenient to store this in a cache to enable rapid access. The URL cache, which is a essential component of the web crawler system is such a cache. Therefore, the URL cache is investigated in dept in section 2.6.

2.1 An introduction to web crawlers

In 2003, the University of Berkley conducted a study of how much new information is created each year. In the study they estimated that the surface of the WWW consists of 167 Terabytes. [5]. In the same study they also claim that the Internet is the fastest growing medium.

To enable regular users to cope which such a large corpus of information, a number of search engine companies have emerged. About 85% of the users on the WWW claims to be using a search engine to find information on the Internet. [18]. The most vital part of the search engine service is the web crawler system.

The web crawler system is utilized by the search engine service to "crawl" a given set of web pages. To crawl a web page is to download the given web page and extract the desired content and URLs linking to new pages. The crawl is performed with the intention of discovering new content on previously visited web pages and to discover new web pages to visit. The desired content, extracted from a given web page, is stored in an external data storage.

It is important for web crawler system to be able to find new content and keep previously retrieved content, stored in the data storage, updated. This is to ensure that relatively new information is available for the search engine service.

The web crawler is provided with one or more entry points. A typical entry point is a main page of a web site or a start page. A start page is normally a collection of URLs, with the intention of providing the user easy access to other popular and useful destinations of the web.

For focused crawling, explained in section 2.3.2, it is normal to use the main page of a web site as the entry point. The main page, in most cases, contain links to all the main areas of the web site. And, therefore, provides a good entry point for crawling the entire web site.

As for the broad web crawling strategy, explained in section 2.3.1, a start page or similar is used as the entry point. A start page will provide links to many parts of the web. The start page, therefore, provides a good entry point for the broad web crawling strategy.

The web crawler will download the entry point web document and extract the desired content from it. This content is stored for later indexing. The crawler will also discover web pages that has not been previously visited. These new web pages is put into a queue, for later processing.

Crawling the web is a continuous and resource demanding task. Due to size and the continuous growth of the web, web crawler systems needs to evolve and find new, efficient, ways to keep up with the growth of the web.

2.2 Architectural overview

There exists numerous architectures for web crawler systems. In [9] a typical high-level architecture is presented, along with a list of published crawler architectures. The paper also propose a new crawling architecture.

Figure 1.1, in chapter 1, shows an example of a typical high-level architecture of a web crawler system, similar to the one found in [9]. Each of the components will briefly be explained in its own subsection below.

2.2.1 Downloader/parser

The downloader/parser component is sometimes referred to as a the web crawler, web spider, web robot or web agent, and can be viewed as the heart of the web crawler system. The main objective of the web crawler is to “crawl” the supplied web page.

To crawl a web page means that the downloader will visit a given URL supplied from the scheduler. The downloaded content will then be cleaned and parsed by the parser, fixing up potential errors and badly nested markup tags. In turn, URLs and other relevant content are extracted from the downloaded content. Finally, the extracted URLs are sent to the queue component, and the content is sent to the storage component.

2.2.2 Storage component

The storage component in a web crawler system is responsible to store the data and meta-information supplied by the crawler.

What is stored in the storage component is dependent on the purpose of the web crawler system, but one of the most common scenarios is to store textural information found on web pages. This information is later indexed, explained in section 2.4, to enable search in the retrieved data.

The storage component can be anything from a large scale database or an in-memory database to a XML file or a text file. This is implementation specific for each system, and depends on requirements for the amount of data and transfer rate the storage component must be able to handle.

2.2.3 Queue

The queue component of the web crawler, contains a collection of URLs to be crawled. The queue is responsible for arranging the URLs in correct order, as decided by the scheduler. Figure 1.1, in chapter 1, illustrates the role of the queue in a web crawler system.

If the so-called URL cache does not contain an extracted URL, the URL will be regarded as new, and will be placed in the queue. If the URL cache contain the extracted URL, the URL have already been added to the queue, and the URL will be discarded.

2.2.4 URL cache

The primary task for the URL cache to ensure that there will not be any duplicated URLs in the queue. If there exists duplicated URLs, this could result in that the crawler will go into a infinite loop, wasting resources on already visited web pages. This will again result in resources spent on already visited web pages and potentially valuable content from other web pages would never be retrieved.

2.2.5 Scheduler

The schedulers main task is to provide the crawler with a URL to crawl, from the queue. The scheduler is also responsible to re-arrange the queue based on how the different URLs are prioritized. Note that there exists a number of different policies of how to prioritize URLs.

2.3 Crawling strategies

Different web crawler systems have different purposes. Some web crawler systems are used solely for discovering new web pages on the web, and some are used for the dodgy business of collecting e-mails for so called spamming. Other web crawler systems is used to continuously monitor a limited domain or area of the web. The purpose of a crawler monitoring the web is, for instance, to check for updates (e.g. check if content of the web page has changed since the last visit).

A web crawler that monitors an area of the web, utilizes a web crawling strategy. In this section we will investigate different web crawler strategies.

2.3.1 Broad web crawling

Broad web crawling is a web crawling strategy utilized to perform large scale crawls on the web. This large scale crawls require a large amount of resources, like bandwidth, memory and storage space. These resources are required to perform as expected, namely to cover as much of the web as possible within a given period of time. [13]

Although, to cover as much of the web as possible, within a limited period of time, are somewhat an extreme approach. However, this approach provides an efficient method for discovering new parts at the web at a high pace. [13]

The broad web crawling strategy emphasize the importance of collecting new individual web pages. For this strategy, discovering new web pages are as important or even more important than a complete coverage of a given web area. [13]

2.3.2 Focused web crawling

When the web crawler system is set to only crawl a limited area of the web, it is referred to as focused crawling. Focused web crawling is defined as a crawl of a small to medium sized collection of web pages, with the criteria of complete coverage of the whole limited area. [13]

A web crawler system using a focused crawling strategy are limited to a specific area of the web. The most common way to limit a given area of the web to crawl for a web crawler system, using a focused crawling strategy, is to limit the crawl to a DNS based web domain. This provides an easy way to filter out all paths from a given web page to areas outside the specified domain.

A practical example of utilizing a focused web crawling strategy on a single web domain could be as follows: Let us say we are set to crawl the well known technology blog Gizmodo. Then the limited area of the crawl would be every page under the domain *gizmodo.com*. This would include every web document situated under the gizmodo.com domain, for instance *http://gizmodo.com/5511678/apple-ipad-review*, which is an iPad review.

2.3.3 Continuous web crawling

In most cases, when utilizing the focused crawler strategy, it is important for the web crawler to keep the limited area up to date. Therefore it is usually combined with continuous crawling.

Continuous crawling is when the web crawler only revisits previously visited areas. The crawler is looking for changes in previous visited areas, it is therefore important to use a strategy that most often visit updated web pages. (e.g. visits those pages that actually are updated since the last visit.)

A practical example of continuous crawling, is to monitor an ongoing discussion on a discussion board. A discussion board allows users to post new comments in a continuous going discussion. After the first visit, the web crawler revisits the discussion board and the crawler will now download new comment on a already discovered discussion.

2.3.4 Revisiting strategies

Since the web grows with a high rate, as stated in section 2.1, it becomes more and more important for commercial data mining companies to be able to refresh their previously fetched content in an efficient manner. To refresh previously fetched content, in the context of web crawlers, means to revisit previously visited areas of the web and fetch all the changes since last visit.

Utilizing a revisiting strategy, provides the advantage of being able to update the stored content from a given, previously visited, web page. The revisiting strategy introduces the term freshness. Freshness indicates how out of date the previously stored content from a given web page is. [10]

A motivation for commercially utilizing revisiting strategies, is to be able to provide a near to real time search functionality. Near real time emphasise that the freshness is very good. In other words, that the given web page gets revisited by the web crawler shortly after changes are made on the web page. An example of changes can be a user posting a comment on a blog.

Another essential factor to enable a good search functionality is indexing techniques.

2.4 Indexing

To provide an end user full-text search functionality for already crawled parts of the web, some kind of processing of the already extracted data must be performed. The most common way of processing the given data is by applying a indexing technique along with document ranking. Figure 2.1 illustrates the role of both the indexer and document ranking in a search engine service. [26]

Indexing is an optimization technique which enables fast search on big collections of data. This data is most often stored in database solutions or as file hierarchies. Indexing increases the speed of storage look-up, allowing the data in the data source to be accessed directly. This is accomplished without any sequential search through the whole data source. [24]

The benefits of utilizing indexing, not surprisingly, comes with a price. The indexing itself is performed as a pre-processing operation. In other words, the indexing techniques require processing of all the data that will be made available by the indexer.

Another significant drawback are the additional overhead of the index data. The index is shown in figure 2.1. In many cases the extra storage space required might be of a considerable size. [24]

If the search engine service is slow, the end user will experience the service as unresponsive and of low quality. This will in worst case result in a great loss of users. If the search engine service provides fast search functionality, on the other hand, the customer will experience the service as responsive and resilient. Therefore both drawbacks, the extra processing and storage space required, can be defended as necessary to provide end users with a good search engine service, even though indexing require more storage space and processing resources.

2.5 Application of web crawler systems

Internet services as search engines, notification, maintenance and analytic services, to mention a few, are all services which utilize a vast amount of information which is located all over the web. To be able to cope such vast amounts of information, a web crawler system is used. Hence, for such services as mentioned above, the web crawler are considered to be the most important component. Without the web crawler, it would not be possible to download and index such vast collection of information from the web.

2.5.1 Search engine systems

A search engine can be viewed as an interface between the end user and the web. Figure 2.1 gives an overview of all the back-end elements in the search engine service, as well as it illustrates the natural order of the elements.

Search engine companies like Yahoo! and Google enables users to search the web to find relevant information, or discover new sources of information. This information is discovered and kept updated by automated web crawler systems.

Figure 2.1 illustrates that the search engine service are divided in two processes, one off-line and one on-line. For each valid web page downloaded by the web crawler in the off-line process, the content retrieved are stored in a storage component, named collection in 2.1. The storage component are briefly explained in section 2.2.2.

The content stored in the storage is also processed by the indexer. Indexing is an optimisation technique applied to increase the search speed as explained in section 2.4.

The on-line process, of the web search engine, contains the actual search functionality. The search functionality, which is most commonly made available for the end user through a graphical web interface, are triggered on request.

2.5.2 Analytic services

Web crawler systems are also used in web analyzing services, which provides tools for analyzing end users activity on the web. Social media is considered as such user activity, and are user generated content in form of conversations and interactions between users on the web.

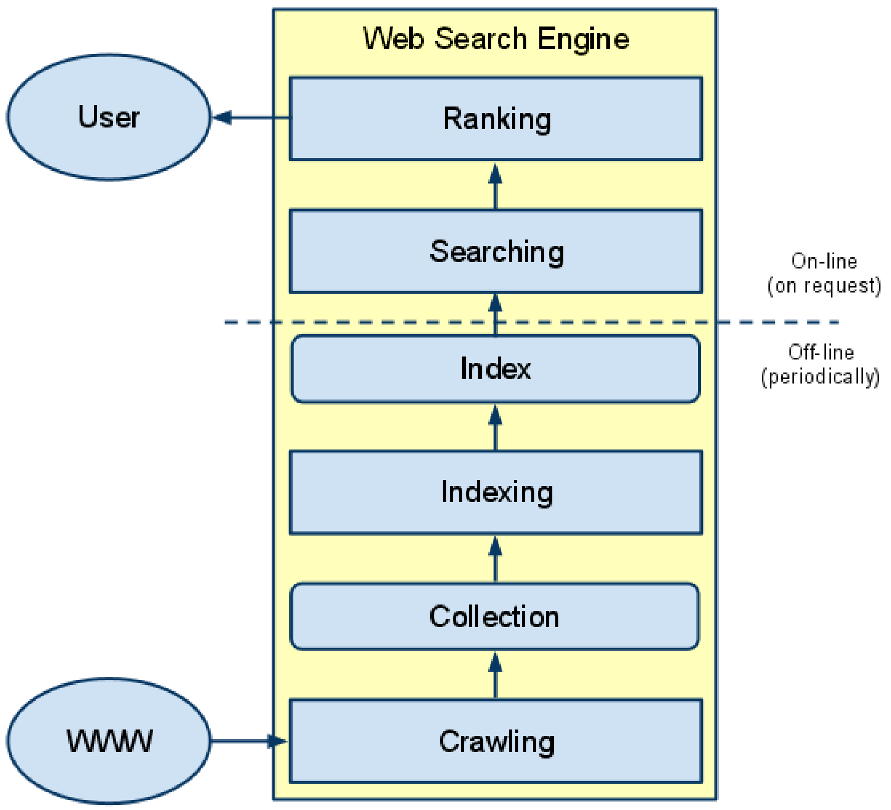

Figure 2.1: This figure illustrates how a search engine service works as an interface connecting the end user to the WWW. [9] The square boxes represents processes and the rounded boxes represents stored data.

User generated content differs from other media channels on the web. The big difference is that it is the readers which generates the content, interact and shares the knowledge, not an author from a corporate news-paper or similar. The users utilize media channels such as Facebook, Youtube, Twitter, different blogs, user review sites and forums to share and interact.

When users discuss a product or a service within a social media channel, it is referred to as word-of-mouth marketing or marketing buzz. This buzz, is a valuable asset for for companies delivering such products and services. In other words, it is an efficient and direct channel for the latter companies to learn what their customers say about their product and services.

Companies like Integrasco and Radian6 utilizes web crawler systems to retrieve this user generated discussions and conduct different types of analysis on it.

Hence, the web crawler is an essential part of analytic services on the web.

2.6 The URL cache

As mentioned in section 1.1.2, this thesis deals with how elements can be cached in a space efficient manner. The URL cache is therefore an important part of this thesis, and need to be properly explained.

To explain the purpose of caching, we try to simplify it by drawing a parallel to the real world. Such a parallel can be drawn between a cache and a woman's handbag. For the same reason as a web crawler system use a cache to store frequently requested elements found on the Internet, does a woman use her handbag to store items that she frequently use. An example of such item can be money. By storing a given amount of money in her handbag, she does not have to walk the extra distance to the bank each time she purchase something from a store.

Since a web crawler system must handle a large amount of URLs, that is accessed each time the crawler visits a new web page, it is convenient to store these URLs in a cache to enable rapid access. The URL cache will store the URL of each visited web page, to ensure that the crawler does not visit the same web page twice in a single crawl. If a crawler is allowed to visit a web page more than once, it is possible for the crawler to go into an infinite loop. Hence, one of the primary goals of the URL cache is to help preventing this from ever happening.

2.6.1 Infinite loop example

Figure 2.2 illustrates an a web crawler going in an infinite loop between two web pages, where the two web pages that is providing a link to each other. Web page A provides a link to web page B, and web page B provides a link to web page A.

Consider the following scenario: The web crawler is set to crawl web page A. The web crawler only discover one link on page A, leading to web page B. When the crawler crawls web page B, a link to A is discovered, and the crawler will then go back to A and think this is a new page.

If not some sort of mechanism is applied to prevent the crawler to go into this infinite loop, the crawler will waste resources and never finish the crawling process. It is therefore important to have some sort of mechanism to prevent this.

As mentioned above, preventing an infinite loop is the main objective of the

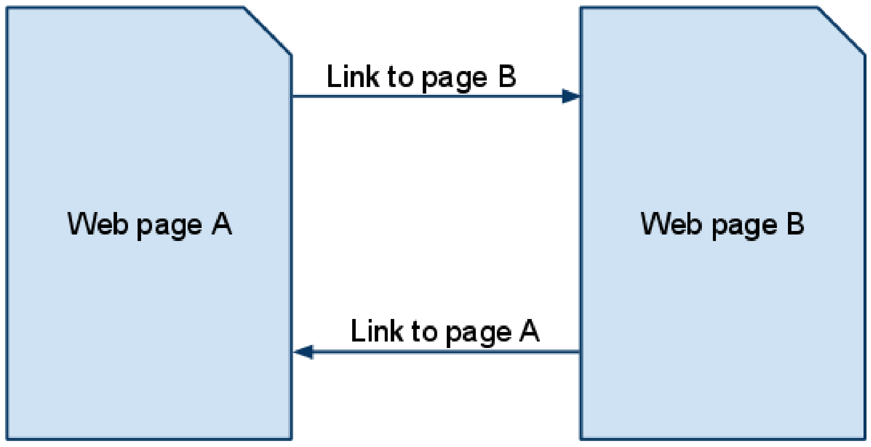

Figure 2.2: This figures shows an example of two web pages linking to each other

URL cache. The URL cache will keep a record of the previously visited URLs. Since a web page can contain large amounts of URLs linking to other web pages, it is important that the URL cache provides a fast way fast to query for an existing URL. In other words, provides a fast way to determine if a URL has been previously visited.

Chapter 3

Bloom Filter

This chapter will focus the space efficient data-structure Bloom filter, due to the importance of Bloom filters for this thesis. Bloom filters are briefly explained in section 1.1.3, while this chapter investigates Bloom filters in dept.

The origin of the Bloom filter, presented in section 3.1, was introduced by Burton H. Bloom in the early 70's. We take a look at what Bloom accomplished and what have happened since then.

The Bloom filter consists of two main elements. One of the elements is a bit array, presented in section 3.2. The bit array is a collection of bits used to store positions.

The other element is a finite number of hash functions. Hash functions are an important part of the Bloom filter, and are therefore thoroughly explained in section 3.3. Here we discuss the area of application and nature of the hash function. The strengths and weaknesses of the hash function, in general, are considered. We especially draw our focus towards hash function collisions. Different non-cryptographic and cryptographic hash functions are also discussed.

A Bloom filter combines the space efficiency of the bit array of size m, with k number of hash functions. This is further explained in section 3.4. This section also investigate disadvantages of the Bloom filter and take a close look at some of the applications of the Bloom filter.

The next section, namely section 3.5, investigates the Bloom filter in-depth. We examine both main parts of the Bloom filter, both the bit array and hash functions, and explain how we add to and query the Bloom filter. The Bloom filter have one major disadvantage, which is the price for its space efficiency, namely the occurrence of false positives. False positives are therefore explained in depth.

In 3.5.2 we explain what a false positive is and why they occur.

3.1 History

In 1970, Burton H. Bloom published the paper “Space/Time Trade-offs in Hash Coding with Allowable Errors” [6]. In [6], new methods to reduce the amount of space required to store hash values in a set are introduced. Bloom considers the trade-offs between the amount of space required and the time to identify a non-member of the given set with an allowable error frequency. The error frequency is later named false positives and the introduced data-structure is later named Bloom filter.

The Bloom filter are commonly used in Databases and Linguistic applications [17], and have been around since the 1970’s. Despite the fact that the Bloom filter have been around for four decades, it is only the last decade or so the filter have received widespread attention. The field of Data Mining is one of the areas the Bloom filter is applied.

A Bloom filter consists of two main components, namely a bit array and a set of hash functions. To understand how these components interact, each component will be thoroughly described in its own section.

3.2 Bit array

In general, an array is defined as a collection of systematically arranged objects. It is similar to the definition of an array in computer science. The major difference is that, in computer science, each object position is identified by a stored value. This value is often referred to as an index or a key.

A bit is the smallest data type in computer science. The bit type can only hold the value 0 or 1, and is the basic type all information in computing is based on. Due to the low level of abstraction, bit operations are cheap to perform and bit values cost almost nothing to store.

Storing a single bit by it self would not accomplish much, but a collection of bits can be utilized to represent data objects, positions and similar. A bit array is a systematically arranged representation of such a collection.

3.2.1 Bit array: Area of application

Bit arrays are are utilized in many areas where space efficiency is important, due to its compact representation. The most common usage of the bit array is to represent simple group of boolean flags or ordered collection of Boolean values. This is often done to create compact or efficient data structures. In the following subsections we present two data structures that utilizes a bit array.

Priority queue

A priority queue is an abstract data type. This data type is characterized by its ability to locate the item with the highest priority in the queue. In other words, the priority queue ranks all its elements, highest to lowest, and pops out the first item in the queue when asking for the item with the highest priority.

Priority queues utilize bit arrays to keep track of which slots in the queue containing an element. In more technical terms, the bit i in the bit array is set to 1 if, and only if, there exists an element in the queue at position i. The bit array utilize a hardware supported operation called “find-first-one”, which provide extremely fast search. In other words, the computational cost used to query a priority list for its top priority element is extremely low, due to the utilization of the bit array.

Bloom filter

Construction of succinct data structures are another area of application for the bit array, where both the compactness of the bit array and the operations available are utilized. The most significant operations are position based look-up and bit count, for instance counting all bits set to 1. The Bloom filter, which is based on the bit array, is such a succinct data-structure. [8]

3.2.2 Advantages and disadvantages

A bit array provides the advantage of storing boolean values in a compact manner. This is much due to the simplicity if the data structure. Since the bit array is essential only an array of bits, the advantage is that it provides fast set and look-up operations, both which have the cost of $O(1)$ [19].

However, despite the advantages and the simplicity that the bit array provide, there are also some disadvantages. The main disadvantage of the bit array is that storing an element without any form of compression would not lead to any gain in time or space, compared to a regular data structure. It would rather lead to a higher cost of resources and be harder to handle. In other words the bit array is not a wise choice when storing data elements without any form of compression.

3.2.3 Bit array in depth

The bit array is an arranged collection of bit elements. As mentioned above, each bit element can only hold the value 0 or 1. Figure 3.1 illustrates a bit array of 8 bit elements and two operations. A set operation which sets the bit in position 4 in the bit array to 1, and a get operation performing a look up on bit element 6 and returns the bit value 1. These two operations are the basic operators of the bit array.

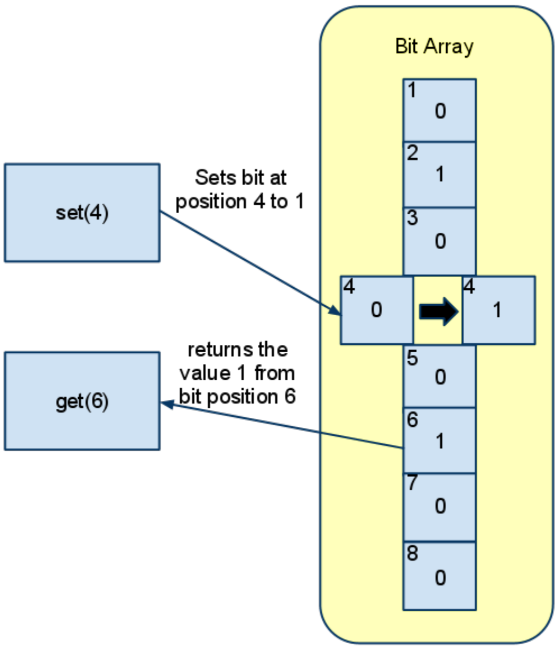
Bit Array
set(4)
Sets bit at position 4 to 1
get(6)
returns the value 1 from bit position 6
1 0
2 1
3 0
4 0
4 1
5 0
6 1
7 0
8 0

Figure 3.1: The figures illustrates a bit array of 8 bits and a set and a get operation. The set operation sets the bit in bit position 4 to 1. The get operation fetches the value 1 from bit position 6.

3.3 Hash functions

A hash function is a mathematical function which is designed to map a collection of data into a fixed length value. This value is referred to as a hash sum or hash value. The hash sum is a value, often represented in hexadecimal or decimal values, within a fixed size. [4]

3.3.1 Hash functions: Area of application

Hash functions are utilized in a variety of areas in computer science. Examples of areas where hash functions are widely used are: Data structures, authentication, authorization and integrity checks for files and messages.

To obtain the integrity of files or messages in a file or message transaction, a hash function can be utilized. The hash function generates a hash value that represents the file content, meaning that if the file content change, the hash value will change. By also transmit the hash value, the transaction recipient are able to check if the integrity of the file or message is maintained. [4]

The reason for ensuring if the integrity of a file or a message are maintained, is to make sure that the transaction is completed without errors like package loss or security issues like so-called man in the middle attack.

3.3.2 Collisions

The big drawback with hash functions are collisions. Collisions occur when 2 or more input elements are mapped to the same hash value.

To illustrate the problem, consider the scenario where a hash function is used to check if a file has been modified. First a hash is generated for the content of the file, then to check if the file has been modified at a later point in time, a new hash is generated. If the two values is equal, the file has not been modified, if the latter value is different, the file has been modified. A collision will happen when the file has changed, but the hash of the content, the hash value, results in the same value. The file will then wrongly be regarded as not changed. [4].

3.3.3 Characteristics of a good hash function

That a hash function is as collision resistant as possible is one of the characteristics of a good hash function. To obtain a good collision resistance, it is important to achieve distributed hash values. In other words, that the hash values are evenly distributed over the value area limited by the hash function.

A hash function of 32 bits have an limited area of less than 2^{32} available value positions. The figures at 3.2 illustrates examples of hash function distribution with an limited area of 2^4 available value positions. Figure 3.2-A illustrates an even distribution of hash values. Figure 3.2-B illustrates an uneven distribution, as all off the bits set to one are centralized in the red area.

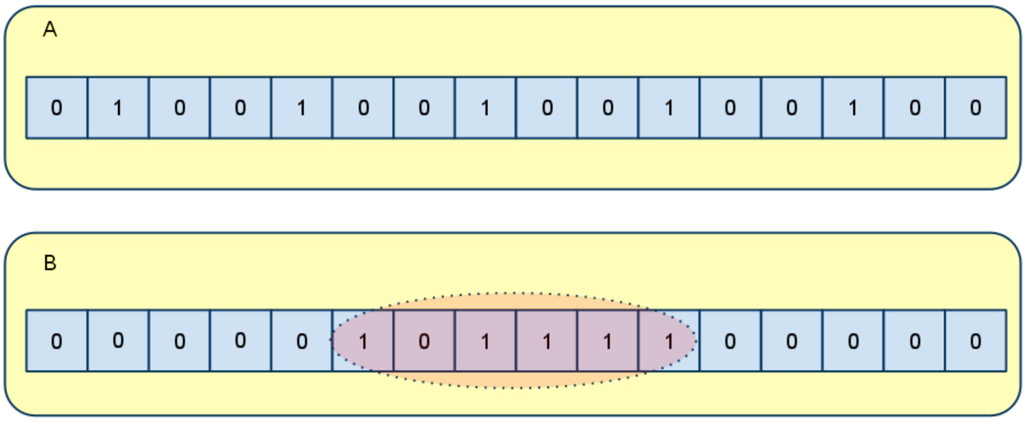

Figure 3.2: The figures illustrates examples even and uneven distribution of hash values. Figure A shows an even distribution and figure B shows an uneven distribution.

If a new element was to be hashed by the same hash function utilized in figure 3.2-B, compared with figure 3.2-A, the probability of the new element would be mapped to the same value as a previous hashed value is regarded as high. That it has high probability of an element being hashed to the same value as another element, implies that it is a high probability of a collision to occur. Therefore, more evenly distributed hash sums leads to a lower probability of collisions.

3.3.4 The hash function: In depth

A hash function is denoted as $h = H(S_i)$ where S_i, of the set S as illustrated in equation 3.2, is a bit string of arbitrary size. This bit string is mapped to the string h witch have a fixed size of n bits. The mapping process of the hash function is described in equation 3.1. [20] [4]

$$H : \{0,1\}^* \rightarrow \{0,1\}^n \tag{3.1}$$

3.3.5 Different hash functions

It is important to choose a hash function that suit the type of input elements you are expecting to obtain even distribution. When choosing a hash function, speed and collision resistance have to be considered.

There are several hash functions which provide good collision resistance. We are going to take a closer look at four of them. The non-cryptographic hash functions Jenkins and MurmurHash, and the cryptographic hash functions MD5 and SHA-1.

The Jenkins hash

The Jenkins hash LOOKUP2 is a non-cryptographic hash function introduced by Robert John Jenkins Jr., also known as Bob Jenkins, back in in 1997. The LOOKUP2 hash was designed for hash table look-up. It is regarded as a fast hash function, due to the relatively low complexity $O(5n + 20)$. [16]

Since the Jenkins hash LOOKUP2 is regarded as a fast hash functions it is also a popular choice for implementing hash tables and Bloom filters. [12].

The MurmurHash

MurmurHash is another non-cryptographic hash function. The hash function is created by Austin Appleby. According to Austin Appleby's website, his Mumurhash2 is able to hash more mb/sec than any other hash functions tested. [3].

The Message-Digest algorithm 5

Message-Digest algorithm 5 (MD5) is a cryptographic hash function, designed by Ron Rivest in 1992. It is a strengthened heir of the MD4 hash function, where one more round is added and each round consists of more operations. [4]. MD4 consists of a fairly simple structure, and it is regarded as a fast hash function. The MD5 hash is somewhat slower than MD4, due to the extra operations added. Still, MD5 is regarded as a fast hash function.

MD5 was intended for digital signature applications [21]. However there exists attacks on MD5 that can be used to generate collisions on MD5. [23]. Even though there exists weaknesses in MD5 it can still be used for non-security purposes, such as hash tables and Bloom filters.

The SHA-1 hash

SHA-1 is another cryptographic hash functions which is also modeled after the MD4 algorithm. [22]. The SHA-1 was developed by the National Institute of Standards and Technology (NIST) and published in 1993 as federal information processing standard, FIPS 180. SHA-1 is a revised version of the original SHA algorithm. A weakness was discovered in the original SHA version, and a revised version was release, this revised version is referred to as SHA-1. [1]

3.4 The purpose of the Bloom filter

A Bloom filter combines the space efficiency of the bit array of size m, with k number of hash functions. The hash functions in the Bloom filter is used to generate position values which corresponds with the range of index values available in the bit array. These position values are utilized to represent a given input element, by setting each bit in the bit array to 1 where the given bit index equals the position value generated.

In other words, when an element is inserted in the Bloom filter, the element is hashed k number of times, one for each hash function, to generate a hash value. The hash value is then used to indicate which bit to set in the bit array.

To check if an element is present in the Bloom filter, the element is hashed k times, one for each hash function, and if, and only if, all the positions in the bit array is set to 1, the element are assumed to be present the filter.

3.4.1 Disadvantages of the Bloom filter and proposed solutions

Even though the Bloom filter is very space efficient, the space efficiency comes with a cost. A Bloom filter may falsely state the an element is present in the set, even though it is not. This is referred to as a false positive, and is thoroughly discussed in section 3.5.2.

A second drawback of the original implementation, described by Burton H. Bloom in [6], is that once an element is inserted in the filter, it is not possible to delete the element from the filter. This is because the positions set in the bit array may also be used to represent another different element that is inserted in the filter. Since the filter uses k number of hash functions there exists a probability that one bit, that is set, is also set by another hash function for another element. The probability of the bit being used by another element will also increase by the number of elements inserted in the filter.

To overcome this drawback Counting Bloom filters have been introduced. A counting Bloom filter uses not a single bit, but a small counter to keep track of the inserted elements. When an element is inserted, the counter will then be incremented, and when an element is removed the counter will then be decremented.

When creating a counting Bloom filter the size of the counter most be taken into consideration. The counter should be sufficiently large to avoid a counter overflow. The drawback of the counting Bloom filter is that it will use more space than a regular Bloom filter, since each counter must be represented by more than 1 bit. If a counter of 4 bits per counter is chosen a counting Bloom filter would use 4 times the space of a regular Bloom filter. [8].

Another drawback of the Bloom filter is that the number of elements that will be inserted in the filter, must be known before the filter is created, to ensure that the probability of a false positive stays below a certain threshold. This is due to the size of the bit array must be known upon creation of the filter.

To be able to dynamically scale a Bloom filter, Scalable Bloom filter was introduced in [2]. A Scalable Bloom filter solves the problem that the number of elements must be known upon creation of the filter. To be able to keep a constant false positive probability, a Scalable Bloom filter will create a new Bloom filter when a certain number of elements has been added to the set. So a Scalable Bloom filter is in effect a series of filters.

3.4.2 Different applications of Bloom filter

Bloom filters have been used in a wide area of places, from databases to web crawler systems and spell checkers.

The paper [14] describes that the Internet Archive crawler implements a test to check if an URL has been visited before. This part of the web crawler uses a Bloom filter to store already visited URLs. By using a Bloom filter to check if an URL already is visited, there exists a chance that a false positive will occur. When a false positive occurs the URL will not be visited, hence the content from the URL will not be retrieved.

In early UNIX systems memory was a limited resources. In some UNIX spell checkers, a Bloom filter was utilized to represent a dictionary to overcome the limitation of little memory. This offered a very compact representation of the dictionary, which was very advantages due to the memory limitation. The problem with this solution was that a false positive in the Bloom filter would result in that a misspelled word being ignored. Hence in some cases a misspelled word would not be corrected. [8].

Bloom filters have also been used in databases to reduce cost of communication. A Bloom filter can be used with distributed databases to reduce the cost of sending data from one database to another, by using a Bloom filter to represent the data. The advantage of this scheme is that it is very cost efficient, in term of communication cost, to send a Bloom filter to represent the data, instead of the data itself. [8].

3.5 Bloom filter: In-depth

A Bloom filter consists of only two main components, namely the bit array of length m, and a collection C_H, of k number of hash functions. In other words a fairly simple data-structure from a high level point of view. The only complex part is the collection, C_H, of hash functions applied. It is therefore important to have a insightful understanding of the application of hash functions in the Bloom filter.

$$S = \{S_1, S_2, \ldots, S_{n-1}, S_n\ \} \tag{3.2}$$

The Bloom filter are used to represent a set of data elements S, which consists of n number of elements, as shown in equation 3.2. These n elements are described by a bit array of m bits, where all bits m are initially set to 0. [17]

It is important to consider the fill ratio of the Bloom filter. The fill ratio is described by m / n, and determines the amount of bit used to represent each element S_1. A higher fill ratio, means more bits to represent a given element. Hence, a higher fill ratio use more memory, but it also reduces the false positive ratio.

3.5.1 Basic operations of the Bloom filter

This section investigates the two requisite operations the Bloom filter needs to support, namely to add an element and query an element. Hence, those are the only two operations needed to serve the purpose as a URL cache.

Adding an element

$$C_H = \{\ H_1, H_2, \ldots, H_{k-1}, H_k\ \} \tag{3.3}$$

The number of hash functions in a Bloom filter is denoted k. For each element n that is inserted in the Bloom filter, k number of bits are set to 1. Hence one bit is set for each of the hash functions utilized by the Bloom filter. In other words multiple hash functions, as described in equation 3.3, are utilized by the Bloom filter to select which bits to set to 1. [17]

Algorithm 1 Adding an element to the Bloom filter

```
1: for all $H$ in $C_H$ do
2:     $h := H(S_i)$
3:     $B_h := 1$
4: end for
```

Algorithm 1 describes the process of adding an element to the Bloom filter. The algorithm loops through each hash function, denoted by H, in the collection of hash functions, denoted by C_H. The generated hash value h of input element S_i is generated by the hash function $H()$. The bit at position B_h in the bit array B is then set to 1.

$$B = \{B_1, B_2, \ldots, B_{m-1}, B_m\} \tag{3.4}$$

Figure 3.3 illustrates how an element is added to a empty Bloom filter. First the element is hashed k times with the k different hash functions. Each hash function

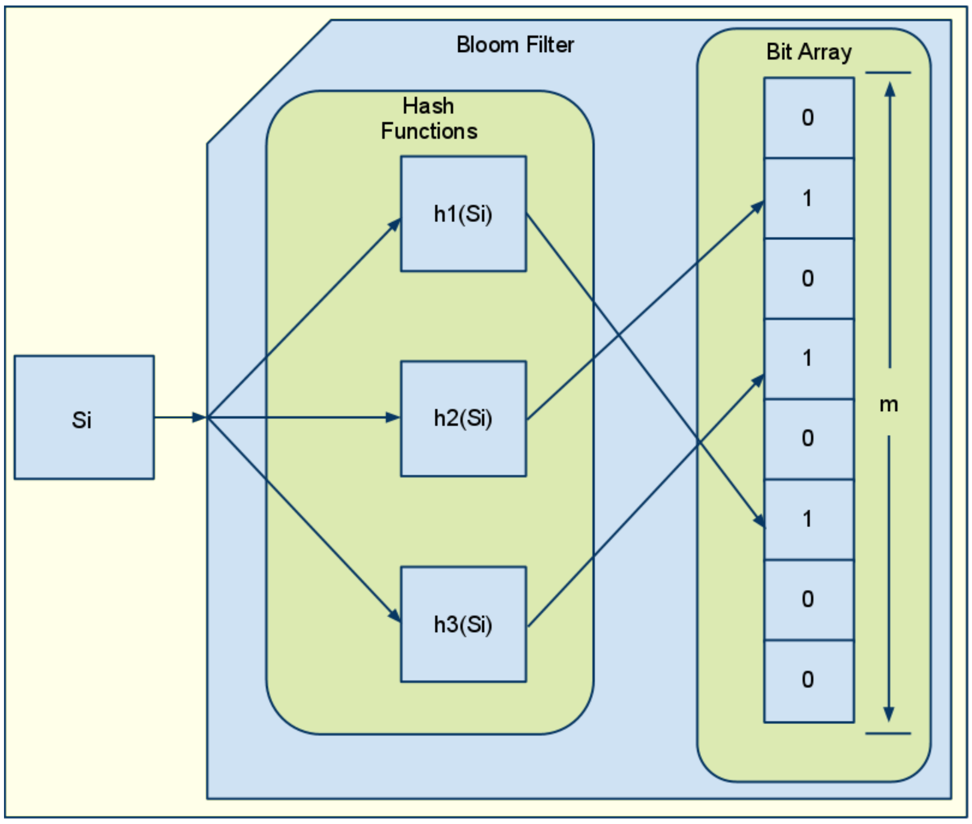

Figure 3.3: The figure illustrates an example of storing element S_i in a Bloom filter. The Bloom filter contains an internal bit array with size m, and uses three hash functions (h_1, h_2 and h_3) to add the given element S_i to the bit array.

will return a valid number in the range 1 to m. This number is used to point to a position in the Bloom filters bit array, which is presented in the equation 3.4. Each bit position pointed to by the number outputs of the hash functions is then set to 1.

If a hash function returns a bit position in the Bloom filter which is already set to 1, the bit value will remain the same. When this happens, it increase the change of a false positive to occur.

There are three different incidents that can result in a hash value equal to a bit position in the bit array that is already set:

1. A collision in the utilized hash function occur, which means that the input

element is mapped to the same hash value with the same hash function a previous input element have been mapped to.

2. The hash function produce the same hash value for the given input element as another hash function have produced for another input element that is already inserted in the Bloom filter.

3. The given element is already added to the Bloom filter.

False positive are explained in depth in section 3.5.2.

Querying the Bloom filter

A query in this context is really an inquiry. Inquiry means to look up or ask for. In other words, a query is a question that returns an answer.

The same hash functions utilized to add elements to the Bloom filter are utilized to query the filter. To query the filter means to ask the filter if it contains a given element. [17]

Figure 3.4 illustrates how to query the Bloom filter to pinpoint if an element, S_i, is present in the filter. The first step is to utilize each of the hash functions to create an hash value of the element. This is performed by each of the k number of hash functions.

Algorithm 2 Query for an element S_i in the Bloom filter

```
1: $b := true$
2: for all $H$ in $C_H$ do
3:     $h := H(S_i)$
4:     if $B_h == 0$ then
5:         $b := false$
6:         break
7:     end if
8: end for
9: return $b$
```

The next step is to check if each of the hash values, where each represents a bit position for the internal bit array in the Bloom filter, points to a bit in the bit array which is set to 1. If all is set to 1, it returns true and false is returned if one or more bits has value 0.

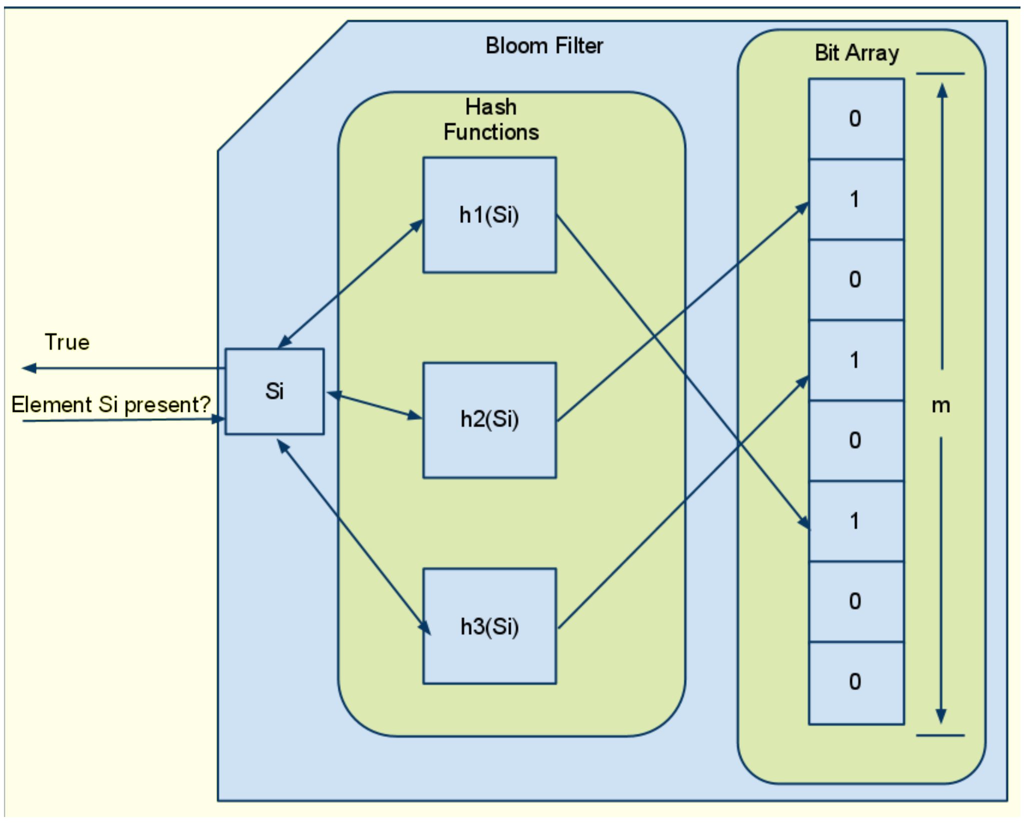

Figure 3.4: The figure illustrates an example of querying a Bloom filter for an element. The bloom filter is asked if element S_i is present in the bit array. The Bloom filter uses three hash functions (h_1, h_2 and h_3), to determine if the element is present or not.

The element S_i is determined not to be present in the Bloom filter if false is returned from the query. Obviously, if true is returned S_i is determined to be present in the filter. The example, in figure 3.4, returns true since all bits looked at where set to 1.

Algorithm 2 demonstrates how to query a Bloom filter for an element S_i. First a variable b, is set to true. This variable is used to indicate if the element is present in the filter. Then all hash functions in the set C_H is looped through. A hash value h is then computed for the element S_i. Then we check if the position, which is the hash value h, is set to 0 in the bit array B. If the position is set to 0, we know that the element is not present in the filter, and we set the variable b to false and breaks the loop. If the position in the bit array is not set to 0, we continue to the next hash function. If all hash functions returns a position, for the element S_i, that is not set to 0, the variable b will remain true, and true will then be returned after all the hash functions is looped through.

3.5.2 False positives

When querying a Bloom filter, there is a certain possibility of receiving a false positive. A false positive can occur based on the two first of the three different incidents mentioned by the enumerated list in the section 3.5.1 under *adding an element*. The last incident, which never happens in practice if the Bloom filter is implemented correctly, would not have any influence on the false positive rate. The reason is that all the bits is already set to one.

To explain the concept of false positives, the following example are given:

Two people, John and Jack, are having a conversation about music. They are discussing how important it is to support the bands they are listening to by paying for their music. John asks Jack if he have bought the newest Slayer album. Jack lies and answer yes.

The fact that Jack states that, him owning the new Slayer album is true, when it is false, makes it a false positive. In other words something stated as true, that is false, is a false positive.

When querying an element, and the incidents mentioned above occurs for each of the k hash functions utilized by the Bloom filter, a false positive occur. In other words, a false positive occur when each of the k hash functions returns positions in the Bloom filter that is set to one, even though the queried element is not present in the filter.

For each new element inserted to the Bloom filter, k of the m bits in the filter are set to one. Logically, the more elements that is inserted in the filter, the higher the probability of a false positive. In other word, because more positions in the bit array is set to 1.

3.5.3 Probability of false positive

In [8] a mathematical formula for calculating the probability of a false positive is given, this formula is shown in equation 3.5. Equation 3.5 assumes that all the hash functions is perfectly random.

$$P = (1 - p)^k = \left(1 - e^{-kn/m}\right)^k \tag{3.5}$$

In equation 3.5 P is the probability of a false positive, n is the number of inserted elements, k is the number of hash functions used and m is the size of the bit array.

In [8] the assumption that all hash functions is perfectly random is made. This is not the case for all hash functions, so in reality the probability will be higher than the one calculated using equation 3.5.

Chapter 4

Proposed Bloom filter strategies

This chapter will outline the proposed solution strategies in detail.

This thesis propose three novel solution strategies, briefly described in section 1.4, each aiming to reduce the false positive rate for a Bloom filter utilized as a URL cache. These strategies was elaborated from the problem statement, presented in section 1.3.

We propose two main solution strategies, a spatial and a temporal strategy, and one combination of the latter two, namely a spatio-temporal strategy. Each of the proposed solution strategies are thoroughly explained in this chapter, each within its own section.

In chapter 2 we investigate web crawler systems. One of the biggest issues concerning web crawlers systems are memory consumption. The URL cache is one of the subareas the issue of memory consumption apply. The URL cache, explained in section 2.6, is used to keep track of all unique URLs visited by a web crawler system.

One way of limiting the memory consumption of the subarea mentioned above, is to apply the space efficient data-structure Bloom filter as URL cache. By utilizing the Bloom filter, which is thoroughly explained in chapter 3, as a URL cache will, presumably, reduce the memory consumption, though, not without a price.

Bloom filters allow a certain error probability, which is referred to as the false positive rate. A false positive, explained in section 3.5.2, occurs when the Bloom filter states that an element is present in the filter, when it is not.

When a Bloom filter is utilized as a URL cache in web crawler system, and a false positive occur, the Bloom filter will state that the URL the filter is queried with have already been visited. Hence, the web page the URL is linking to will never be visited.

That a false positive leads to that a web page never being visited, leads to that potential valuable content never will be extracted from the given web page. Hence, false positives leads to loss of valuable content for a web crawling system utilizing a Bloom filter as URL cache.

Base on the latter assumptions, of false positives leading to loss of, presumably, valuable web content for a web crawler system utilizing a Bloom filter as URL cache, we propose three novel URL caching strategies based on temporal and spatial approaches. These strategies, which are briefly explained in section 1.4 in the introduction chapter are as follows:

1. **A temporal strategy for reducing false positives.**
2. **A spatial strategy for reducing false positives.**
3. **A spatio-temporal strategy for reducing false positives.**

Each of the three strategies proposed will be thoroughly explained in the sections below, but first the additional sub research question introduced in 1.5 is thoroughly explained.

4.1 Seeded hash functions

As stated in 3.4, the Bloom filter utilize k number of different hash functions to represent an element in the filter. To use k number of different hash functions can lead to a practical problem, namely a lack of enough different suitable hash functions. Also, an assumption were made, that it would be better to utilize the best suitable hash function k times, than k different hash function. A sub-research question was therefore added to list of research questions, see section 1.5.

The idea behind the sub-research question, is, presumably, to benefit from both utilizing the most fit hash function, and to remove the problem with lack of suitable hash functions. The proposed solution is to use a single hash function k times, seeding the input URL with a different salt each time. The preferred result would be that the Bloom filter utilizing the different salts as seed instead of different hash functions would produce close to the same false positive rate, or better.

Hash functions are an important part of the Bloom filter and are thoroughly explained in section 3.3. To decide which hash function that is most fit for our solution, 4 different hash functions, all briefly explained in section 3.3.5, will be evaluated in a Bloom filter setting.

If the preferable result is accomplished, all other test cases presented in chapter 5 will utilize Bloom filters with the best fit hash function with different salts as seed, instead of different hash functions.

4.2 Temporal strategy

This section will thoroughly explain the novel temporal strategy proposed by this thesis. Temporal can be defined as follows:

If you describe processes or strategies as temporal, you are referring to how they change or endure over a period of time.

Temporal in the context of this thesis is related to a web crawler system applying two web crawling strategies. These strategies, focused and revisiting crawling, both explained in section 2.3, are utilized to, as mentioned above, monitor a limited area of the web over time. Hence, the temporal strategy, as the name might reveal, is based on time.

The focused web crawler strategy is defined as a so-called crawl sequence of a small to medium sized collection of web pages, as stated in section 2.3.2. A crawl sequence can be defined as the whole sequence of visits a web crawler performs, visiting each web page once.

Consider the following example: If we were set to perform a crawl, where the web crawler utilizing a focused crawling strategy, limited to the domain of *gizmodo.com*, a crawl sequence would be defined as the sequence of visits the web crawler had to perform to cover all available web pages under the *gizmodo.com* domain.

The revisiting strategy is based on revisiting a set of previously crawled web pages. The purpose of the revisiting strategy is to discover new content on the previously visited web pages upon revisit.

The intention of the temporal strategy, is to reduce the false positive rate for a web crawler system utilizing a Bloom filter as a URL cache. The strategy will, presumably, limit the false positive rate by taking advantage of the characteristics of a web crawler utilizing revisiting and focused crawler strategies. Hence, the

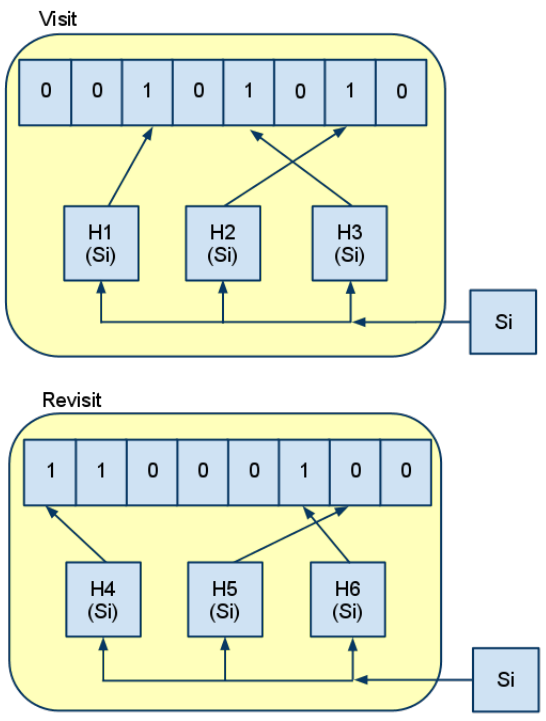

Figure 4.1: Illustration of adding an element, S_i, on visit and revisit utilizing the temporal strategy, where each Bloom filter have three hash functions.

application of the temporal strategy on the latter web crawling system, are from here referred to as the temporal solution.

The temporal solution utilize a single bloom filter. The bloom filter contains k number of hash functions, or k number of salts. When the temporal solution perform a crawl, false positives may occur. Further, the idea of the temporal strategy is to use a new Bloom filter containing another combination of hash functions or salts upon a revisit. This will, presumably, provoke false positives on different URLs upon revisit. Hence, over a time-span of two crawls, the false positive rate, arguably, will be reduced compared to a single crawl.

In other words, the intention of the temporal solution is to provoke false positives on other URLs on the the second crawl, compared to the first crawl, as illustrated in figure 4.1. In other words, the figure 4.1 illustrates a scenario where element S_i is added to a Bloom filter on both visit and revisit. Further, the hash functions utilized by the filter in both visit and revisit maps S_i to a to a bit position in the filter, which is set to 1.

From figure 4.1, we can see that the combination of bits set to 1 in the Bloom filter for each visit and revisit, are different from each other. By obtaining this difference, which is the main goal of the temporal strategy, will presumably lead to other coalitions between hash functions on revisit compared to visit. Therefore, the false positive rate will, arguably, be reduced from an overall perspective, basing this assumption on that different hash function collisions gives different false positives.

To consider another scenario, where the temporal strategy is applied in a web crawler system utilizing a Bloom filter as URL cache, and the element S_i represents an URL linking to a specific web page. A Bloom filter is queried two different times, asking if the URL S_i is present in the filter, one time for each crawl. At the first crawl the query return that S_i is present in the Bloom filter. This leads to that the web page S_i is linked to, will not be visited. At the second crawl the query return that S_i is not present in the set, and the web page S_i is linked to, gets visited. Hence, this indicates that the query from the first crawl returned a false positive, and even though the web page did not get visited on the first crawl, it was crawled on the next.

It is worth noticing, that each of the crawls on its own, will not reduce the false positive rate. Alone, they will rather act as a standard single Bloom filter solution. Although, over a time-span of several crawls, the false positive rate will, arguably, be reduced.

The algorithms 1 and 2, describing the operations of adding and querying a Bloom filter, presented and explained in section 3.5.1, are the same as the Bloom filter on each visit uses in the temporal strategy. Therefore, the description made of each of the operations, in the section 3.5.1, apply for the temporal strategy and will not be described again here.

The temporal solution may introduce a time delay on some content that the crawler visits, since a false positive in the filter will result in that the URL will be visited upon a later crawl.

The worst case scenario for the temporal solution is that some URLs will be hashed to the same values each time, resulting in that every time the filter is queried for the URL a false positive will be given, and the URL will never be visited, which leads to the content of that page will never be retrieved.

4.3 Spatial strategy

This section will thoroughly explain the novel spatial strategy proposed by this thesis. Spatial can be defined as follows:

Spatial is used to describe things relating to size, space, area, or position; a formal word.

Spatial, in the context of this thesis related to memory consumption.

The spatial strategy utilizes a combination multiple Bloom filters to, arguably, reduce the false positive rate. Each of the Bloom filters contains a k number of hash functions, or a single hash function utilized k number of times, seeded with different salts each time.

When combining multiple filters, it is important that each filter is different. For a combination of two Bloom filters, if both filters are equal, a false positive in the first filter will also be a false positive in the second filter. Hence, the false positive probability will stay the same, and the memory consumption would be approximately the double, compared to a single filter. To ensure that the filters are not equal, different hash functions or a single hash function where the input elements seeded with different salts will be used to create each filter.

The spatial strategy is based on knowledge of; a Bloom filter will never return a false negative when queried. In other words, a Bloom filter returning that an element is not present in the filter, when queried, returns a correct answer. This means that it is certain that the queried element is not present in the filter.

Further, the idea for the proposed spatial strategy is to take advantage of the fact that the Bloom filter never return a false negative. Since the Bloom filter does not return false negatives, we assume that, as long as at least one of the filters in a multiple Bloom filter solution returns a negative answer, the element queried for is not present in the filter. Therefore, when querying a multiple Bloom filter solution for an element, and an ambiguous answer is returned, all filters which returns a positive answer will be regarded as false positives.

As described in section 3.5.1, the Bloom filter only needs two basic operations, add and query. Figure 4.2 shows an example of how element S_i is added into two filters. The filters is configured with 3 hash functions each. If we compare which bits that are set in the two arrays, we see that different positions is set in the arrays. This is the goal of the spatial solution. When the values are different, a false positive in the first filter may not result in a false positive in the second filter, assuming no false positive occur in at the same bit position in the second filter.

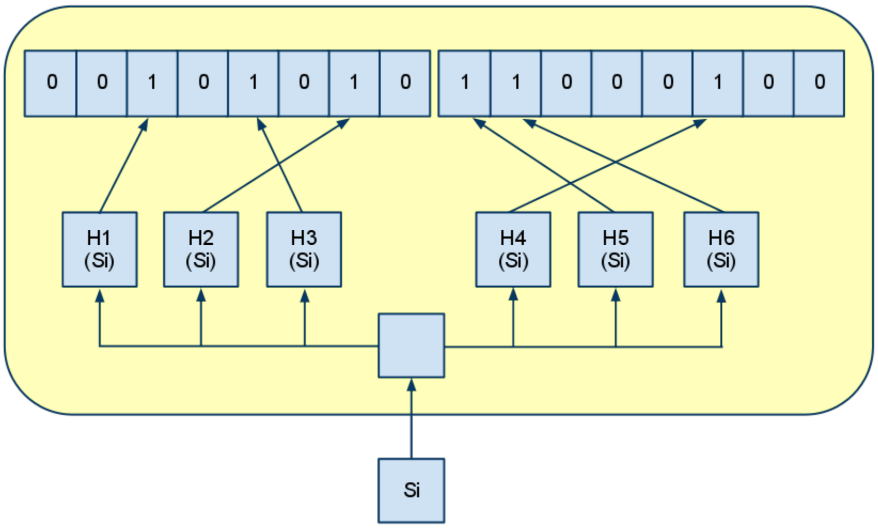

Figure 4.2: Illustration of adding an element, S_i, to two Bloom filters utilizing the spatial strategy, where each Bloom filter have three hash functions.

To clarify this even more, consider the following scenario: For a combination of two Bloom filters, a false positive occurs in filter 1, as a result of the occurrences of collisions in the hash functions belonging to the latter filter. If another set of hash functions is utilized by filter 2, the elements will be located in different positions. Hence a false positive in filter 1 may not be a false positive in filter 2. Therefore, this will, presumably lead to a reduction of the false positive ratio, compared to a single filter with the same amount of hash functions.

The spatial solution is likely to consume approximately the double amount of memory as a single Bloom filter, however, the spatial solution will still utilize less memory than traditional data structures that is used for the same purpose. Hence, due to the compact representation the Bloom filter provide, it should be possible to reduce the false positive rate, and still consume only a fraction of the memory a traditional data structure like for instance a hash set would consume.

The two basic operations of the Bloom filter, explained in section 3.5.1, namely adding an element and querying for an element need to be supported. Algorithm 3 describes how the implementation of adding an URL will work, and algorithm 4 explains how the implementation of querying for an URL will work.

Algorithm 3 describes how to add an element S_i to multiple Bloom filters. For each Bloom filter F in S_f, S_i is hashed by all hash functions H in C_H. Each

Algorithm 3 Add an element S_i to multiple Bloom filters

```
1: for all $F$ in $S_f$ do
2:     for all $H$ in $C_H$ do
3:         $h := H(S_i)$
4:         $B_h := 1$
5:     end for
6: end for
```

hash value h generated by $H(S_i)$, B_h is set to 1. In other words, the hash value h returned by $H(S_i)$ represents a bit position in the given Bloom filter, which is set to 1.

Algorithm 4 Query for an element S_i in multiple Bloom filters

```
1: $b := true$
2: for all $F$ in $S_f$ do
3:     for all $H$ in $C_H$ do
4:         $h := H(S_i)$
5:         if $B_h == 0$ then
6:             $b := false$
7:             break
8:         end if
9:     end for
10: end for
11: return $b$
```

Algorithm 4 demonstrates how to query multiple Bloom filters for a given element, S_i. First the collection of all filters S_f is looped through. Then for each filter, all the hash functions is looped through at line 3. For each hash function the element S_i is hashed. Then, based on the bit position values, created by each of the hash functions, a look-up is performed on the filter. The look-up is performed to check if the bit position values hashed for S_i is present in the filter. If bit value 1 is returned, the variable b will still be true, if bit value 0 is returned, the variable b is set to false and the algorithm will exit, returning false. This is performed for each hash function on each filter. If the bit value 1 is returned for each hash function in each Bloom filter, true is returned.

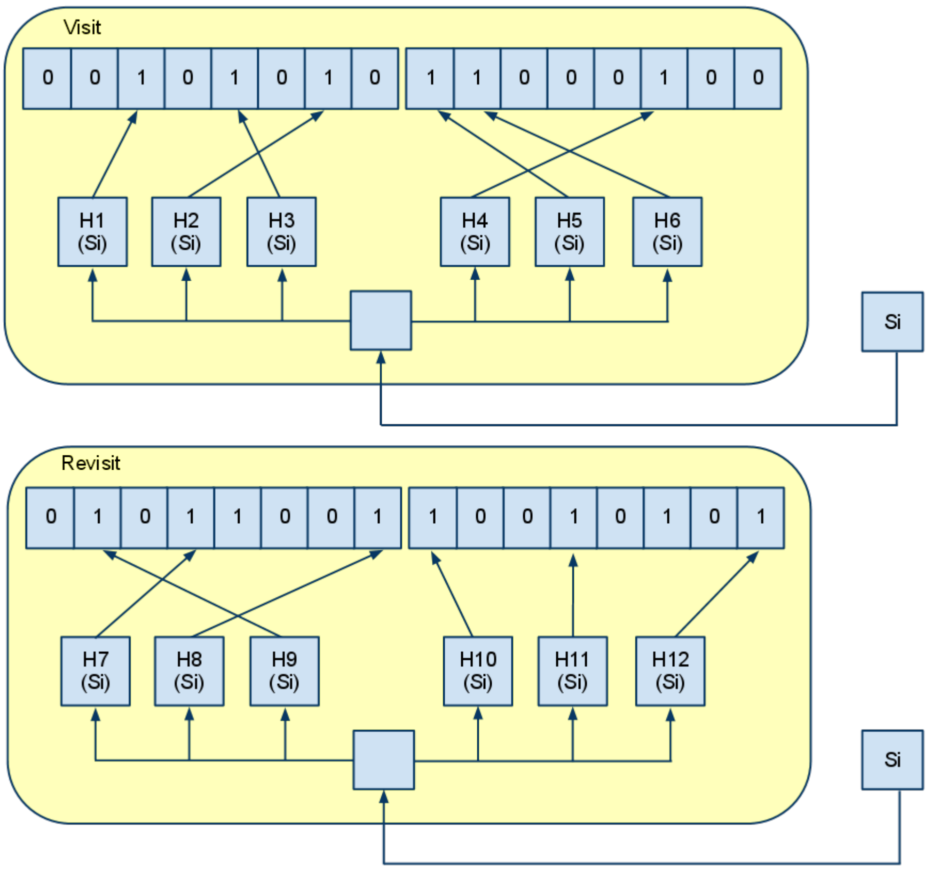

Figure 4.3: Illustration of adding an element, S_i, to two Bloom filters on visit and revisit, utilizing the spatio-temporal strategy, where each Bloom filter have three hash functions.

4.4 Spatio-Temporal strategy

This section will thoroughly explain the novel spatio-temporal strategy proposed by this thesis. Spatio-temporal can be defined as follows:

Spatio-temporal is used to describe the relation to to size, space, area, or position, referring to how they change or endure over a period of time.

The two latter sections have thoroughly described the novel temporal and spatial strategies proposed by this thesis. The novel spatio-temporal strategy is, as you may have guessed, a combination of the two latter strategies. The intention of combining the temporal and spatial strategies are to utilize the reduction of false positives these strategies, presumably, provide to further reduce the false positive rate.

The figure 4.3 illustrates an example of how an element, S_i, is added into two Bloom filters both on visit and revisit when utilizing the spatio-temporal strategy. Each Bloom filter in both visit and revisit utilize three hash functions each. If we compare which bits that are set in the two filters in visit and the two filters in revisit, we observe that all the filters contains different combinations of bits set to 1.

Obtaining different combinations of bits set to 1 in each of the filter, will increase the probability of discovering a false positive and presumably decrease the false positive rate. This is the intention of the spatio-temporal strategy. When the values are different, a false positive in the first filter may not result in a false positive in the second filter, and a false positive in occurring in visit may not result in a false positive in revisit.

The algorithms 3 and 4, describing the operations of adding and querying multiple Bloom filters, presented and explained in section 4.3, are the same as the combination of multiple Bloom filters which each visit uses in the spatio-temporal strategy. Therefore, the description made of each of the operations, in the section 4.3, apply for the temporal strategy and will not be described again here.

Chapter 5

Validation and Testing

In this chapter we test the strategies proposed and thoroughly explained in chapter 4. We first describe the test data used, and then explain how the different test cases is conducted and what the purposes of the different test cases is. Each of the results for the different strategies is presented in its own section.

5.1 Test data

To provide correct results it is important to have correct test data. Since our focus is the URL cache in a web crawler system, the test data used will be URLs.

To conduct the different tests we will need 1 000 000 unique URLs to insert in the filter, and 1 000 000 more unique URLs to query with. Hence, the total number of URLs will be a minimum of 2 000 000 unique URLs. Therefore, we will harvest a test set with a minimum of 2 000 000 unique URLs.

To prevent falsely identification of false positives, which can happen if the set of input URLs contains duplicates, the implementation will discover duplicates, and discard the duplicated URLs.

To retrieve the needed test data, an web crawler was implemented in the Python programming language. This crawler was used to retrieve the necessary number of 2 000 000 unique URLs.

5.2 Different test approaches

We will test our solution strategies with two approaches, incremental and non-incremental. This will be performed with the intention of gaining a broader understanding of the test results.

5.2.1 Incremental approach

The incremental approach will simulate the behavior of a web crawler system. Since the solution strategies proposed are placed in a web crawler context, we will mimic the behavior of a web crawler to obtain the most accurate test results.

To simulate the behavior of a web crawler, the incremental approach will start with an empty Bloom filter. Then the filter will be queried if a URL is contained in the filter. After the query, the URL will be inserted, and the same procedure will be repeated for a new URL. This is repeated for 1 000 000 unique URLs. If the filter claims to contain any of the URLs, this will be regarded as a false positive.

5.2.2 Non-incremental approach

The non-incremental test approach will provide worst case test results. When initializing a Bloom filter, the size of expected input elements must be defined. In other words, the filter have a maximum element limit. This limit defines how many elements which can be represented in the Bloom filter without exceeding an acceptable false positive rate. Therefore, we will test our solution strategies with the non-incremental approach to measure the false positive rate for a filled up Bloom filter.

To conduct the non-incremental test, we will first create a Bloom filter, and then insert 1 000 000 unique URLs. After the URLs is inserted, we will then query the Bloom filter with a different set of 1 000 000 unique URLs. If the filter claims to contain any of the latter URLs, it should be regarded as a false positive.

5.3 Experimental setup

Before presenting the analysis and results, we will describe the experimental setup. For each test result presented in this chapter, a test is performed 100 times, and an average is calculated based on the test results from each of the 100 test results. This is performed to ensure that a single test run does not influence the overall results to much.

Due to the fact that the size of the Bloom filter is determined by the number of bits available in the filter, not considering the small overhead created by the hash functions and the bit array, the Bloom filter was configured with different number of hash functions and different bit-size multipliers. To calculate the size of the filter m, equation 5.1 is utilized.

$$m = n * k * M \tag{5.1}$$

In equation 5.1, m is the bit size of the Bloom filter, n is the number of URLs that will be inserted, k is the number of hash functions and M is the bit-size multiplier. The bit-size multiplier determines the number of bits available for each bit inserted in the Bloom filter.

Considering the issue that a hash function is a fairly complex function with somewhat high computational time in the context of data-structures, a higher number of hash functions in the Bloom filter would mean a higher overall computational time. Therefore, different bit-size multipliers was used to observe if a smaller number of hash functions combined with a higher bit-size multiplier would produce near to the same results as a larger amount of hash functions and a smaller bit-size multiplier without consuming a lot more memory.

The number of hash functions used, in the different test cases, range from 1 to 10. The bit-size multiplier that will be tested is the following: 1.7, 2.0, 2.5, 3.0, 4.0 and 5.0. This will provide a wide range of test results which should be sufficient enough to indicate if it is better to increase the bit size multiplier or add more hash functions.

All the different tests is conducted on the same computer. The computer is configured with an AMD Athlon II X4 620 processor, with 4 cores and running at 2.6GHz. The computer is also configured with 4 GB of RAM.

5.4 Hash function approaches

The results presented in this section will have two closely connected purposes. The first purpose is the provide an answer to the sub research question presented in section 1.5 in the introduction chapter. The research question asks if seeding URLs with k different salts, only using the best suited hash function will provide

1 HF	FP(%)
Jenkins	21.307%
MurmurHash	21.309%
MD5	21.321%
SHA-1	21.332%

Table 5.1: Average false positive count for 1 hash functions for a incremental run.

close to the same results as using the k best suited hash functions. Seeded hash functions thoroughly explained in section 4.1.

The other purpose is to determine, which of the selected hash functions is the best suited one. The hash functions tested is: Jenkins, MurmurHash, MD5 and SHA-1. MD5 and SHA-1 is cryptographic hash functions, that tends to be slower than normal hash functions, since they are used for cryptographic purposes. Cryptographic hash functions tends to be slower since it should not be able to compute the original value from the hashed value, as explained in section 3.3. Jenkins and MurmurHash is general purpose hash functions and therefore does not need to have this property. Hence it is faster to compute the hash value of an element with Jenkins and MurmurHash.

To test the false positive of the different solutions we will use two test scenarios: Incremental and non-incremental, as explained in section 5.2.

5.4.1 Incremental

To test the false positive rate with the incremental test, will start with an empty Bloom filter. First an URL is fetched and the filter will then be queried if the URL is present in the filter. If the filter claims that the URL is present, a false positive is discovered, due to the fact that the URL has not yet been added to the filter. After querying the filter, the URL is added to the filter, and a new URL is fetched and the process is repeated. This is done with 1 000 000 unique URLs, and each test is repeated 100 times and an average is calculated based on the 100 runs.

The incremental test is used to simulate how a web crawler would actually interact with the URL cache when crawling a set of web pages. This test will therefore simulate the behaviour of how URLs would be added to the URL cache in a real web crawler environment. The results from this test will indicate how well a Bloom filter would perform if used in a URL cache.

Table 5.1 shows the results of four incremental tests that are run with one hash

2 HF	FP(%)
Jenkish And MurmurHash	5.829%
Jenkins (seeded)	5.824%
MurmurHash (seeded)	5.832%
MD5 (seeded)	5.824%
SHA-1 (seeded)	5.823%

Table 5.2: Two hash functions, incremental

function. All results in table 5.1 is tested with a Bloom filter that has the bit array size set to m = 2 000 000.

The test results for one hash function Bloom filter, presented in table 5.1 indicates that with one Bloom filter utilizing one hash function the false positive probability(FP) will be close to equal for all four Bloom filters. The filter which utilizes Jenkins produces the best result, but the deviation between the four are so small that it would be regarded as equally good.

All Bloom filters produce a false positive rate close to 21.3%, which would result in that a web crawler would miss 21.3% of the ideal visited web pages. Hence, due to the large number of false positives, a Bloom filter should not be used with only one hash function or the bit-size multiplier should be increased.

The test results for two hash function Bloom filter, presented in table 5.2, show that the test results for the four Bloom filters utilizing the seeded hash functions produce approximately the same false positive rate. The filter utilizing SHA-1 is slightly better then the others, though, the deviation between the latter filter and the other seeded bloom filters are again so small that it would be regarded as equally good.

However, two interesting discoveries are made. The first is that false positive rate have gone from approximately 21.3% to 5.8%. This implies that by doubling m from 2 000 000 bits to 4 000 000 bits, the false positive rate is diminished to almost a fourth of the rate seen in the table 5.1.

The other discovery made, are that the Bloom filters utilizing seeded hash functions produce just as low false positive rate as the filter utilizing a combination of two different hash functions. This result seems promising for the seeded hash function research question, even though, at this point it is to early to draw a conclusion.

The test results for three hash function Bloom filter, presented in table 5.3, show that the four Bloom filters, utilizing the seeded hash functions, produce

3 HF	FP(%)
Jenkish, MurmurHash and SHA-1	1.765%
Jenkins (seeded)	1.762%
MurmurHash (seeded)	1.760%
MD5 (seeded)	1.763%
SHA-1 (seeded)	1.760%

Table 5.3: Three hash functions, incremental

	1 HF	2 HF	3 HF
Jenkins	21.307%	5.824%	1.762%
MurmurHash	21.309%	5.832%	1.760%
MD5	21.321%	5.824%	1.763%
SHA-1	21.332%	5.823%	1.760%

Table 5.4: Average false positive count for an incremental test run with seeded hash functions.

approximately the same false positive rate. When we compare the results from this table with the latter results, presented in table 5.1 and table 5.2, we observe a repeating trend.

This trend imply that the four different hash function performs almost equally good when utilized by a Bloom filter using the incremental approach. We notice that there are some minor differences, SHA-1 and Jenkins seems to perform best. Although, as already mentioned, the deviation between the results are to small to conclude with that one are better than the others.

There are also two other tendencies to observe when comparing the latter table of test results with table 5.2. One tendency is that the false positive rate does not decrease linearly with the number hash functions, k, or number of bits, m, in the Bloom filter. It seems rather to converge against a number close to, or even, zero.

The other tendency discovered, when comparing the results from latter table, 5.3, with the results from table 5.2, is that is that the Bloom filters, utilizing seeded hash functions, produce just as low false positive rate as the filter utilizing a combination of different hash functions. Hence, this confirms the sub research question concerning seeded hash functions, presented in section 1.5 of the introduction chapter, for Bloom filters when tested with a incremental approach.

Table 5.4 provides an overview of all test results from table 5.2 and 5.3, where the Bloom filter utilizes seeded hash functions. Since the deviation between the

1 HF	FP(%)
Jenkins	39.365%
MurmurHash	39.364%
MD5	39.354%
SHA1	39.330%

Table 5.5: Average false positive count for 1 hash functions (HF) in a non-incremental run.

different results are so small, they are regarded as equally good when utilized, by a Bloom filter acting as a URL cache, to reduce the false positive rate, when tested with a incremental approach.

5.4.2 Non-incremental

To conduct the false positive test with the non-incremental approach will first add 1 000 000 unique URLs to an empty Bloom filter. After inserting the 1 000 000 URLs, another set of 1 000 000 unique URLs is used to query the filter with. Since the URLs used to query the filter with is not stored in the filter, a claim from the filter that it contains any of these URLs will be a false positive. Each test is performed 100 times and an average is calculated based on the results.

Table 5.5 shows the results of a non-incremental test that is run with one hash function. All results in table 5.5 is tested with a Bloom filter that has the bit array size set to m = 2 000 000.

The test results for one hash function Bloom filter, presented in table 5.5 indicates that with one Bloom filter utilizing one hash function the false positive probability (FP) will be close to equal for all four Bloom filters. The filter which utilizes SHA-1 produces the best result, but the deviation between the four are so small that it would be regarded as equally good.

All Bloom filters produce a false positive rate just above 39.3%, which would result in that a web crawler would miss around 39.3% of the visited web pages. Hence, due to the large number of false positives, a Bloom filter should not be used with only one hash function or the bit-size multiplier should be increased.

The test results for two hash function Bloom filter, presented in table 5.6, show that the test results for the four Bloom filters utilizing the seeded hash functions produce approximately the same false positive rate. The filter utilizing SHA-1 is slightly better once more, the deviation between the latter filter and the other

2 HF	FP(%)
Jenkins And MurmurHash	15.492%
Jenkins (seeded)	15.470%
MurmurHash (seeded)	15.497%
MD5 (seeded)	15.463%
SHA-1 (seeded)	15.456%

Table 5.6: Average false positive count for 2 hash functions in a non-incremental run.

3 HF	FP(%)
MurmurHash, Jenkins & SHA-1	6.093%
Jenkins (seeded)	6.104%
MurmurHash (seeded)	6.092%
MD5 (seeded)	6.098%
SHA-1 (seeded)	6.091%

Table 5.7: Average false positive count for 3 hash functions in a non-incremental run.

seeded bloom filters are again so small that it would be regarded as equally good.

However, two interesting discoveries are made. The first is that false positive rate have gone from approximately 39.3% to 15.4%. This implies that by doubling m from 2 000 000 bits to 4 000 000 bits, the false positive rate is diminished to less than half of the rate seen in the table 5.5.

The other discovery made, are that the Bloom filters utilizing seeded hash functions produce just as low false positive rate as the filter utilizing a combination of two different hash functions. This result seems promising for the seeded hash function research question, even though, at this point it is to early to draw a conclusion.

The test results for three hash function Bloom filter, presented in table 5.7, show that the four Bloom filters, utilizing the seeded hash functions, produce approximately the same false positive rate. When we compare the results from this table with the latter results, presented in table 5.5 and table 5.6, we observe a repeating trend.

This trend imply that the four different hash function performs almost equally good when utilized by a Bloom filter using the non-incremental approach. We notice that are some minor differences, mainly that SHA-1 performs best each

	2 HF	3 HF
Jenkins	15.470%	6.104%
MurmurHash	15.496%	6.092%
MD5	15.463%	6.098%
SHA-1	15.456%	6.091%

Table 5.8: Average false positive count for a non-incremental test run with seeded hash functions.

time. Although, as already mentioned, the deviation between the results are to small to conclude with that one are better than the others.

There are also two other tendencies to observe when comparing the latter table of test results with table 5.6. One tendency is that the false positive rate does not decrease linearly with the number hash functions, k, or number of bits, m, in the Bloom filter. It seems rather to converge against a number close to, or even, zero.

The other tendency discovered, when comparing the results from latter table, 5.7, with the results from table 5.6, is that is that the Bloom filters utilizing seeded hash functions produce just as low false positive rate as the filter utilizing a combination of different hash functions. Hence, this confirms the sub research question concerning seeded hash functions, presented in section 1.5 of the introduction chapter, for Bloom filters when tested with a non-incremental approach.

Table 5.8 provides an overview of all test results from table 5.6 and 5.7, where the Bloom filter utilizes seeded hash functions. Since the deviation between the different results are so small, they are regarded as equally good when utilized, by a Bloom filter acting as a URL cache, to reduce the false positive rate, when tested with a non-incremental approach.

Table 5.9 shows the test for 1 Bloom filter which only uses seeded hash functions. The hash functions used is the Jenkins hash, and all the URL is seeded with an unique value before the hash value is computed. The left column denotes the number of seeds, or hash functions (HF), and the top row denotes the bit-size multiplier (M).

5.4.3 Hash function selection

The main purpose of this section was to provide an answer to the sub research question, concerning seeded hash functions, presented in 1.5. Another, closely related part of this section was to select a hash function to utilize the seeding on.

	1 Bloom filter					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	8.7927%	6.1041%	3.5882%	2.2770%	1.0788%	0.5964%
4 HF	3.9137%	2.3985%	1.1832%	0.6458%	0.2395%	0.1098%
5 HF	1.7396%	0.9418%	0.3910%	0.1824%	0.0534%	0.0194%
6 HF	0.7730%	0.3716%	0.1281%	0.0519%	0.0118%	0.0035%
7 HF	0.3455%	0.1468%	0.0419%	0.0147%	0.0027%	0.0006%
8 HF	0.1528%	0.0573%	0.0142%	0.0043%	0.0006%	0.0001%
9 HF	0.0679%	0.0225%	0.0046%	0.0012%	0.0001%	0.000025%
10 HF	0.0301%	0.0090%	0.0015%	0.0004%	0.00004%	0.000004%

Table 5.9: Non-incremental Bloom filter query for 1 Bloom filter with seeded hash.

We have tested seeding of hash functions with four different hash functions, for a Bloom filters utilizing both two and three hash functions. These tests have been performed both with an incremental and non-incremental test approaches. For each time test of the four seeded hash functions have been tested with two or three hash function Bloom filters, both by incremental and non-incremental approach, the same test have been performed on a Bloom filter utilizing the same number of different hash function. This is performed with the intention of comparing the seeded hash Bloom filters with the Bloom filter utilizing different hash functions.

The results for each test, both in incremental and non-incremental, provide us with the same result, namely that the seeded hash function Bloom filters provide just as low false positive rate as a Bloom filter utilizing different hash functions. Therefore, seeded hash function based Bloom filters will be utilized in for all the following tests in this chapter.

To be able to utilize this seeded hash function approach, we also need a hash function to apply the approach on. The different hash functions tested was Jenkins, MurmurHash, MD5 and SHA-1.

We were not able to draw a conclusion on way or another, based on the false positive rate presented by the test results in this section, due to that the test results show that each of the hash functions nearly the same false positive rates. Therefore, we have to base our decision on other factors.

Another factor to take in to consideration is processing time. Since the MD5 and SHA-1 is cryptographically hash functions, as explained in 3.3, due to security reasons, they requires more processing time than Jenkins and MurmurHash.

Hence, we discard MD5 and SHA-1, and need to choose between Jenkins and MurmurHash.

We consider Jenkins and MurmurHash to be as good as equal, but Jenkins have provided, somewhat, slightly lower false positive rate for the temporal strategy tests. Since the thesis is closely related to web crawler systems, we choose the hash function that performed best in the incremental test results. Therefore, the Jenkins hash will be utilized as a seeded hash function, by the Bloom filters in the rest of this chapter.

5.5 Memory consumption

To measure the memory consumption we will compare the different configurations of our Bloom filter implementation, against the ArrayList data structure found in java.util.Arraylist, and HashSet found in java.util.HashSet. The two data structures was chosen, due to their suitable characteristics, where both have the needed functionality to function as a URL cache in a web crawler system, namely *add* and *contains*.

To measure the memory consumption of the Bloom filter, HashSet and ArrayList, in turn, 1 000 000 unique URLs was added to each and the memory consumption of each data structure was measured. The memory consumption was measured utilizing an resource profiling agent.

This agent is based on Java Instrumentation Application Programming Interface (API) [15]. The Instrumentation API have methods to measure memory used by an object, in a running application. Direct usage of Java Instrumentation only measures the memory usage of a single object, and therefore an agent which also measure the object references is utilized.

The memory measured is the memory that is consumed by the instance of the data structure. To give an accurate indication of how much memory the different data structures will consume, we will measure deep memory consumption. Deep memory consumption is how much memory an object and the references found in that object uses. So all the references from the given instance will also be measured, to given an indication of how much memory would actually be used.

Table 5.10, shows the memory consumption in kilobytes for a single Bloom filter with different number of hash functions and bit-size multiplier. HF is number of hash functions and M is the bit-size multiplier.

From the table, we observe that the more hash functions that is used, the more

	1 Bloom filter					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	660	770	953	1136	1503	1869
4 HF	868	1014	1258	1503	1991	2479
5 HF	1075	1259	1564	1869	2479	3090
6 HF	1283	1503	1869	2235	2968	3700
7 HF	1491	1747	2174	2602	3456	4311
8 HF	1698	1991	2048	2968	3944	4921
9 HF	1906	2235	2785	3334	4433	5531
10 HF	2114	2480	3090	3700	4921	6142

Table 5.10: Memory consumption in kilobytes for one Bloom filter

	2 Bloom filters					
	1.7M	2.0M	2.5M	3.0M	4.0M	5.0M
3 HF	1286	1506	1872	2239	2971	3703
4 HF	1702	1994	2483	2971	3948	4924
5 HF	2117	2483	3093	3703	4924	6145
6 HF	2532	2971	3703	4436	5901	7366
7 HF	2947	3459	4314	5168	6877	8586
8 HF	3362	3948	4924	5901	7854	9807
9 HF	3777	4436	5535	6633	8830	11028
10 HF	4192	4924	6145	7366	9807	12248

Table 5.11: Memory consumption in kilobytes for 2 Bloom filters with different configuration.

memory will be consumed. The memory consumption will also increase by increasing the size of the bit-size multiplier, since this directly affects the size of the bit array, which is essentially the only part of the filter consuming any space.

Table 5.11 and 5.12 summaries the memory consumption for two Bloom filters and three Bloom filters. Both tables shows if the bit-size multiplier and/or number of hash functions is increased, the amount of memory consumed will also increase.

The reason that the memory consumption increases when adding a hash functions or increasing the bit-size multiplier, is that these parameters directly affects the size of the bit array. The size of the bit array is calculated using equation 5.1.

Comparing table 5.10, 5.11 and 5.12 shows that the memory will increase

	3 Bloom filters					
	1.7M	2.0M	2.5M	3.0M	4.0M	5.0M
3 HF	1909	2239	2788	3338	4436	5535
4 HF	2532	2971	3704	4436	5901	7366
5 HF	3154	3704	4619	5535	7366	9197
6 HF	3777	4436	5535	6633	8831	11028
7 HF	4400	5169	6450	7732	10296	12859
8 HF	5022	5901	7366	8831	11760	14690
9 HF	5645	6633	8281	9929	13225	16521
10 HF	6267	7366	9197	11028	14690	18352

Table 5.12: Memory consumption in kilobytes for 3 Bloom filters with different configuration.

Collection	Memory consumed
ArrayList	194560
HashSet	250880

Table 5.13: Total amount of memory consumed by an ArrayList and a HashSet

approximately linear, based on the number of filters that is added. E.g. If one filter is used and you increase to two filters, the amount of memory consumed will increase by a factor of 2.

Table 5.13 summaries how much memory is used by an ArrayList and a HashSet. The ArrayList uses 194560 kilobytes or 190 MB of memory, while the HashSet uses 250880 kilobytes or 245 Mb of memory. The ArrayList and HashSet will increase in size based on the amount of elements that is stored in the structure.

The Bloom filter consumes significantly less memory than the ArrayList and HashSet. The reason that the Bloom filter uses so less memory than the ArrayList and HashSet, is that the only reference the Bloom filter have, is a reference to a bit array. This means that the only thing that consumes memory in the Bloom filter, is the bit array of size m.

The ArrayList and the HashSet must keep references to all inserted elements. Since the URLs is represented as a String, the ArrayList and HashSet must keep references to all the inserted Strings. This explains the much larger memory consumption by the ArrayList and HashSet.

Since our focus is on the URL cache it is not important to have the ability to

Collection	Time in milliseconds
ArrayList	58
HashSet	428

Table 5.14: Time to add 1 000 000 URLs to collection

fetch a URL from a position in the list, which is possible with both the ArrayList and HashSet. The URL cache only needs the ability to query if the URL is visited, hence the ability to fetch an element from a position is considered not necessary.

The Bloom filter is able to query if an element is present, and add new elements. This is the only properties that is needed in the context of an URL cache, therefore an ArrayList and HashSet that stores each URL as a String object will contribute to the overhead, hence explaining the significantly more memory consumed by these collections.

5.6 Performance test

The performance tests is designed to test how well the different data structures does against each other. The goal is to asses that the Bloom filter will not be a bottleneck in the URL cache of a web crawler system.

5.6.1 Add test

The add test is performed to see how much time it takes to add elements to the data structure. The add test uses 1 000 000 unique URLs, and the test is done 100 times, and an average is calculated.

From table 5.14 we see that the ArrayList is the fastest data structure tested, it uses approximately 58 milliseconds to add 1 000 000 URLs. The ArrayList is a very simple data structure and does only insert the next element after the previous inserted element.

The HashSet is a bit slower than the ArrayList and uses approximately 428 milliseconds to insert 1 000 000 URLs. The reason that the HashSet is slower, is due to that it first must hash the value that will be inserted, to generate an unique key.

In the Appendix A.1.3 the results of the insertion time for the different con-

Collection	Time in milliseconds
ArrayList	22000000
HashSet	176

Table 5.15: Time to query for 1 000 000 URLs in collection

figurations of the Bloom filter is shown. A Bloom filter utilizing a combination of three seeded hash functions, uses about 3213 milliseconds to add 1 000 000 URLs. This is approximately 7.5 times the time of a HashSet. The obvious reason that the Bloom filter is slower than the HashSet, is to the number of hash functions utilized. The HashSet only utilized one hash function, while in this case the Bloom filter will use three. By increasing the number of hash functions to four, the time to add 1 000 000 URLs to the filter will increase to 4578. This indicates that the time will increase linearly with the number of hash functions.

Since the Bloom filter uses more time to insert an element it may be more feasible to increase the bit-size multiplier than by adding more hash functions.

5.6.2 Query test

To test the time to query the different data structures, 1 000 000 URL will be inserted, and then we will query for 1 000 000 URLs that is not contained in the data structure. So the time reflected in this test is a worst-case scenario.

Table 5.15 shows the result for querying for 1 000 000 URLs on an ArrayList and a HashSet. It is worth noticing that the time for the ArrayList is calculated. To calculate the time for the ArrayList, 1 000 000 URLs was inserted and it was then queried with an URL that was not contained in the ArrayList. This was done 100 times, and an average was calculated.

From the table we see that the ArrayList is very slow to query. Since we query with an element that is not in the list, all elements must be checked. To conduct the test with the ArrayList with 1 000 000 URLs and then query with another 1 000 000 URLs is not feasible within the time frame of this thesis. The test of the ArrayList is therefore conducted with inserting 1 000 000 URLs, and then doing 100 independent queries and calculating the average. The average for the ArrayList is 22 milliseconds, so the time to query for 1 000 000, not inserted elements in an ArrayList, would be 22 * 1 000 000 = 22 000 000 milliseconds. The query time is due to that there is no look-up mechanism in the ArrayList, each element have to be checked to ensure that the queried element is not inserted.

Since the ArrayList is so slow it is not feasible to use it in an URL cache. If an ArrayList is used as a URL cache and the majority of the URLs that was queried for was not in the set, the ArrayList would become a bottleneck in the URL cache.

From table 5.15, we see that the HashSet is relative fast and uses approximately 176 milliseconds. The HashSet does only need to hash the element once to check if the element is contained. This is the strength of the HashSet since you only have to do one hash to check if an element is present or not.

In Appendix A.1.2, we see the average time to query the different configurations of Bloom filters. The filter performs better than the ArrayList, since the Bloom filter will have a constant look-up time. However the amount of hash functions influence the look-up time. The test indicates that if it is possible, the bit-size multiplier (M) should be increased, instead of adding more hash functions. This is due to that adding more hash functions will increase the insertion time and query time. Still the Bloom filter will use less time than the ArrayList, but not faster than the HashSet. This is because the HashSet only have to do one hash, while the Bloom filter will have to hash the element multiple times.

However, the Bloom filter may use more time to query for elements than the HashSet, but the Bloom filter uses significant much less memory than the HashSet. This shows that if memory is a concern, a Bloom filter is very suitable replacement for the HashSet as an URL cache.

5.7 Temporal strategy

The novel temporal strategy is thoroughly explained in section 4.2. The intention of the temporal strategy is to reduce the false positive rate introduced by the Bloom filter, when utilized as an URL cache in a web crawler system. The temporal strategy will, presumably, reduce the false positive rate compared to a single Bloom filter, by applying a revisiting strategy explained in 2.3.4, adopted from the filed of web crawlers.

5.7.1 Incremental

To be able to test the temporal strategy with an incremental approach, we first inserted 1 000 000 URLs in one Bloom filter, utilizing a set of seeded hash functions, and then record which URL yields a false positive. Further, a new filter was created, and the same URLs is inserted in the new filter, but this time, we applied

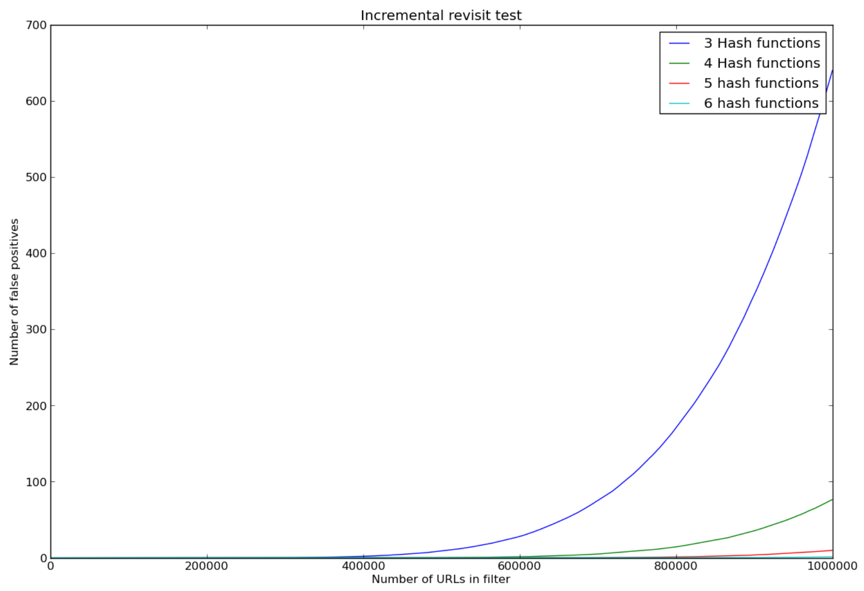

Figure 5.1: Incremental revisit results

different seeds to the hash function. Again which URL yields a false positive will be recorded.

The false positive rate was measured by comparing the false positives from both recorded sessions, and counted all URLs that yielded a false positive in both both sessions.

This test simulates how a web crawler would behave. The crawler would first visits a web page and download the content, and then at a later point in time, revisits the page to check for updates.

The incremental revisit is tested with filters that have 3, 4, 5 and 6 different seeds. This means that each element will be hashed 3, 4, 5 and 6 different times for the different tests. The bit-size multiplier is set to 2.0.

Figure 5.1 shows the results for the incremental revisiting strategy with 3, 4, 5 and 6 hash functions. With three hash functions the overall false positive rate is around 0.064%. This would mean that with a filter that 1 000 000 URL will be stored in, would miss around 640 web pages.

By adding a fourth hash function, the false positive rate is reduced and ends up on around 0.0077% false positives, which is slightly better. Although, a relative high number considering that in a web crawler system, this would result in 77 web pages not being downloaded. Increasing the number of hash functions to 5, results in that the number of false positives is reduced to about 0.001% false positives, and by increasing the number of hash functions to 6, the false positive rate are reduced to respectable 0.00012%.

A we can see from the graph, 5.1, the temporal strategy with an incremental approach is able to reduce the false positive rate significantly, compared to a single Bloom filter, utilizing an incremental approach, shown in the graph A.2.

5.7.2 Non-incremental

To conduct the non-incremental test on the temporal strategy, we first add 1 000 000 unique URLs to an empty Bloom filter. After the URLs is added, we query the filter with another set of 1 000 000 unique URLs, and record which URLs yields a false positive. We then create a new filter, using different seeds, and insert 1 000 000 URLs. We then query with the same set of URLs that was used to query the first filter, and record which URL yields a false positive. We then compare the false positives from both runs, and a URL which yields a false positive in both runs, will be a false positive in the temporal strategy.

Table 5.16 shows the test result non-incremental revisit strategy. Since this test is non-incremental it will show a worst case scenario, meaning that this represents how the revisits strategy would work on a full filter.

What is interesting with table 5.16 is the result for the configuration with 7 hash functions and 4.0 bit Multiplier. Both the preceding test cases yields a false positive probability of 0.0%, while in this case it is 0.0000001%. This is because no matter how many hash functions or how large bit-size multiplier, there is always a chance that a URL will yield a false positive both times it is added to the filter. The probability will decrease when adding more hash functions and increasing bit-size multiplier.

By using 8 hash functions and a bit-size on 2.0 it is possible to achieve a false-positive rate at 0.3%. Or by increasing the bit-size to 4.0 with 5 hash functions, we will achieve approximately the same false positive probability. By increasing the bit-size to 3.0 with 8 hash functions we will reduce the false positive to 0%.

If we compare the results in table 5.16 with the results in table 5.9 we see that for a single Bloom filter using 3 hash functions without the revisit strategy would yield a false positive probability of 8.7927%. On the other hand if we apply the revisit strategy the probability of a false positive drops to 0.7687% which is a sig-

	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	0.7687%	0.3723%	0.1285%	0.0520%	0.0115%	0.0036%
4 HF	0.1519%	0.0574%	0.0139%	0.0043%	0.0006%	0.0001%
5 HF	0.0302%	0.0090%	0.0016%	0.0003%	0.00003%	0.0%
6 HF	0.0060%	0.0014%	0.0002%	0.00002%	0.0%	0.0%
7 HF	0.00113%	0.00022%	0.00002%	0.0%	0.000001%	0.0%
8 HF	0.0002%	0.00003%	0.000005%	0.0%	0.0%	0.0%
9 HF	0.00004%	0.000008%	0.0%	0.0%	0.0%	0.0%
10 HF	0.00001%	0.000001%	0.0%	0.0%	0.0%	0.0%

Table 5.16: Non-incremental revisit filter query for 1 Bloom filter.

nificant improvement. The other configurations also shows a similar improvement in the false positive probability when applying the revisiting strategy.

5.8 Spatial strategy

The spatial strategy test will combine multiple Bloom filters and exploit the fact that Bloom filters does not return false negatives. The novel spatial strategy is thoroughly explained in section 4.3.

To test the spatial strategy we split the tests into two types: incremental and non-incremental.

To be able to test the spatial strategy with an incremental approach, we first start with a combination of multiple Bloom filters with no elements inserted. We then query all the filters to check if an URL is present in the filters. After the query, the URL will be added, and the same will be performed with another URL. This is done for with 1 000 000 unique URLs, 100 times on each test.

To test the spatial strategy we will also conduct a non-incremental test. The non-incremental test will first fill up all the filters with 1 000 000 URLs, and then query all the filters with another 1 000 000 unique URLs. Since all the 2 000 000 URLs are unique, if the filters claim to have any of the URLs used to query with, this is regarded as a false positive.

	1 Bloom filter		2 Bloom filters		3 Bloom filters	
	1.7 M	2.0 M	1.7 M	2.0 M	1.7 M	2.0 M
3 HF	2.6071%	1.7686%	0.1368%	0.0641%	0.0086%	0.0028%
4 HF	0.9492%	0.5656%	0.0214%	0.0076%	0.0006%	0.0001%
5 HF	0.3565%	0.1875%	0.0036%	0.0010%	0.00004%	0.00001%
6 HF	0.1374%	0.0641%	0.0007%	0.0001%	0.000003%	0.0%

Table 5.17: Incremental multiple Bloom filter query.

5.8.1 Incremental

The incremental Bloom filter test will first query each filter if the URL is contained, if all filters claim to have the URL stored in it, the results will be regarded as a false positive. After querying all the filters, the URL is then added to all filters, and then the sequence is repeated.

In appendix A.1.1 we see the graphs for the different tested configurations. The graphs indicates how the false positive probability develops over time. The Bloom filter was tested with 1, 2 and 3 filters with 1.7 and 2.0 bit-size multiplier. All the graphs indicates that using three hash functions will give a relatively high false positive probability after approximately 600 000 URLs is inserted.

Table 5.17 shows the results for the incremental tests. In this table we have also included the false positive probability for 1 Bloom filter, for comparison.

By comparing the results for 1 and 2 Bloom filters in the latter table, for 3 hash functions and 2 bit-size multiplier, we can observe an significant reduction of false positives. The only drawback, except for that the false positive rate is not zero, is that the 2 Bloom filter solution consumes double amount of memory compared to the solution with one filter.

Another discovery is made in table 5.17. If we compare the latter multiple Bloom filter test results utilizing 3 hash functions and a bit-size multiplier of 2 against the single Bloom filter test results with 6 hash functions and 2 in bit-size multiplier, we discover something quite interesting. Both test results have 6 hash functions, and does also get the same false positive rate, namely, 0.0641%.

Further, to try to confirm our suspicion, we compare the test results for; the combination of 2 Bloom filters utilizing 6 hash functions each and have a bit-size multiplier of 2.0 and the combination of 3 filters, 4 hash functions each and a bit-size multiplier of 2.0. Where both have the same false positive rate, 0.0001% and both utilize 12 hash functions. Hence, this imply that it does not matter if two Bloom filters with 6 hash functions and a bit-size multiplier of 2.0 are used or three Bloom filters with 4 hash functions and 2.0 in bit-size multiplier used.

These tests imply that, by comparing a single filter with a multiple filter solution, where both have the same number of hash functions and bit-size multipliers, a reduction of the false positive rate will be achieved. Although, to achieve this, a certain increase of memory consumption must be allowed, namely, the single Bloom filter size multiplied by the number of filters in the multiple filter utilized. Hence, to obtain a reduction of false positives in a spatial Bloom filter solution applying the incremental test approach, a increase of the memory consumption must be allowed.

Even though a reduction of false positives was obtained, a single filter utilizing the same amount of hash functions, and uses the same bit-size multiplier will obtain the same result.

5.8.2 Non-Incremental

To test the spatial strategy with a non-incremental test approach, first insert 1 000 000 URLs in each filter. Further, the URL is inserted all the filters is queried with another 1 000 000 unique URLs. If all the filters claim that any of the URLs used to query with is contained, a false positive will occur.

Table 5.18 show the results for two Bloom filters. By comparing the results for one filter in table 5.9 with the results 5.18 we observe the same as in the incremental test, the false positive probability is reduced.

However if we compare one filter with 6 hash functions with two filter and 3 hash functions. There is actually a slight increase in the false positive probability from 0.7730% to 0.7736%. With the non-incremental test there is still a indication that using multiple filters have minimal effect on the probability of a false positive, when the filters is compared to a single filter with the same total amount of hash functions.

Table 5.19 shows the same test result only with three Bloom filters. If we compare 1 Bloom filter with 9 hash functions, this is the same amount of hash function as: 3 filter * 3 hash function = 9 hash functions total. By comparing 1 filter with 9 hash functions with 3 filters with 3 hash functions we see that there is still no significant reduction in the probability of a false positive.

The other configurations shows similar results for the non-incremental test. Hence non-incremental test does, as well as the incremental test, indicates that there is no significant gain in adding extra filters, instead of hash functions.

	2 Bloom filters					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	0.7736%	0.3727%	0.1283%	0.0522%	0.0116%	0.0035%
4 HF	0.1535%	0.0576%	0.0139%	0.0042%	0.0006%	0.0001%
5 HF	0.0302%	0.0089%	0.0015%	0.0003%	0.000022%	0.000008%
6 HF	0.0060%	0.0014%	0.0002%	0.00003%	0.000001	0.0%
7 HF	0.0012%	0.0002%	0.00002%	0.000001%	0.0%	0.0%
8 HF	0.0003%	0.00002%	0.000004%	0.0%	0.0%	0.0%
9 HF	0.00002%	0.000001%	0.0%	0.0%	0.0%	0.0%
10 HF	0.000004%	0.000001%	0.0%	0.0%	0.0%	0.0%

Table 5.18: None-incremental multiple Bloom filter query for 2 Bloom filters.

	3 Bloom filters					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	0.0680%	0.0230%	0.0047%	0.0011%	0.0001%	0.000022%
4 HF	0.0059%	0.0014%	0.0002%	0.000023%	0.0%	0.0%
5 HF	0.0005%	0.00008%	0.000005%	0.000001%	0.0%	0.0%
6 HF	0.00004%	0.00001%	0.0%	0.0%	0.0%	0.0%
7 HF	0.000008%	0.000001%	0.0%	0.0%	0.0%	0.0%
8 HF	0.000002%	0.0%	0.0%	0.0%	0.0%	0.0%
9 HF	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
10 HF	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 5.19: None-incremental multiple Bloom filter query for 3 Bloom filters.

	2 Bloom filters			
	1.7 M	2.0 M	2.5 M	3.0 M
2HF	0.1534%	0.0577%	0.0142%	0.0042%
3HF	0.0059%	0.0014%	0.00017.%	0.00003%
4HF	0.0002%	0.00004%	0.000002%	0.0%

Table 5.20: Two Bloom filters spatio-temporal solution

5.9 Spatio-temporal solution

These test combine the spatial solution and the temporal solution. The numbers represented in the tables is the probability of a URLs that would not be visited by combining the spatial and temporal solution. Each test is run 100 times and an average is calculated for all the runs.

Table 5.20 shows the results for two filters that uses the spatio-temporal solution. From the table we see that the false positive probability is reduced.

However by comparing the results we can see that a spatio-temporal solution using two filter with bit-size multiplier of 1.7 and 2 hash functions can be compared to one Bloom filter that uses 4 hash functions. (Since 2 filter * 2 hash functions = 4 hash functions total). By comparing the results in table 5.20 with the revisit results in table 5.16 we see that the spatio-temporal solution doest not decrease the probability of a false positive anymore than the temporal solution does.

Comparing the results in table 5.21 with the results in the revisit table 5.16 also shows that there is no significant difference between these tests.

The spatio-temporal tests shows that it is possible to reduce the false positive probability. However if the solution is compared to Bloom filter with the same total amount of hash functions, there is little or no gain by combining multiple filters, with the spatial strategy.

However the the temporal strategy will reduce the false positive probability. The results in table 5.20 and 5.21 indicates that it is only the temporal strategy that is reducing the false positive probability.

	3 Bloom filters			
	1.7 M	2.0 M	2.5 M	3.0 M
2HF	0.0060%	0.0013%	0.00017%	0.00003%
3HF	0.00005%	0.000002%	0.0%	0.0%
4HF	0.0%	0.0%	0.0%	0.0%

Table 5.21: Three Bloom filters spatio-temporal solution

Chapter 6

Discussion

In the previous chapter we tested both the spatial and temporal strategies, and combined the two strategies in a simulated web crawler environment. The tests conducted with the spatial strategy indicates that there is no significant reduction in the false positive probability, by combining multiple Bloom filters. However, the temporal strategy shows that is it possible to reduce the probability of a false positive by swapping the values used to seed the URLs upon a revisit. During this chapter we will elaborate our finding.

6.1 Bloom filter performance

Since the Bloom filter is a space efficient data structure, it is very suitable to be used in a web crawler environment, which must be able to handle a large amount of URLs. Section 5.5 in the previous chapter indicates that there is significant reduction in the memory consumption by utilizing a Bloom filter, compared to other relevant data structures.

The Bloom filter supports two basic operations, add and contains, which is the only operations that is required when the filter is used as a URL cache in a web crawler environment. Since the crawler is only interested if the URL is visited before or not. The data structure used as an URL cache does not need to support the ability to retrieve the specific URL.

The Bloom filter was compared to a ArrayList and a HashSet. The drawback with these two data structures is that they use much more memory than a Bloom filter. The reason for the memory usage is that it is possible to retrieve the inserted element, which is not possible in the Bloom filter. Since it is possible to retrieve the inserted element, the data structures must also store the inserted element in its full representation. This is the reason that both the ArrayList and HashSet uses so much more memory than the Bloom filter.

Since the URL cache does not require to retrieve the full representation of the inserted element, only check if the element is inserted, the Bloom filter is quite suitable to be used in web crawler environment, and offer significant reduction in memory usage.

The query test in section 5.6.2 shows that the ArrayList is very slow to query for an element that is not inserted in the structure, hence the ArrayList should not be used in a URL cache. If the majority of the URLs that the URL cache is queried for is not inserted in the URL cache, the ArrayList may become a major bottleneck for a web crawler system.

The HashSet however is a very efficient data structure to use as an URL cache when querying for elements. This is because the HashSet only operates with one hash functions, that is relative fast. However the HashSet consumes a lot more memory, compared to a Bloom filter. This could become a bottleneck in a web crawler system that must handle a large amount of URLs.

The Bloom filter is slower to query and add elements to, however the Bloom filter uses significant less memory than the other structures tested. So if the small increase in query and insertion time is acceptable, the Bloom filter will yield a significant reduction in memory consumption.

By adding hash functions to the Bloom filter it will become slower, since there more hash values must be computed, both when adding and querying for elements. The test in section 5.6.1 and 5.6.2 indicates that it is more feasible to add a larger bit-size multiplier than to add more hash functions. The false positive probability is decreased when adding more hash functions or by increasing the bit-size multiplier.

However adding more hash functions causes the filter to be slower, since more hash values must be computed, both when adding and querying. Increasing the bit-size multiplier however does not affect performance that much. Hence using a higher bit-size multiplier will decrease the false positive probability, while keeping the filter relative fast when adding and querying for elements.

6.2 Seeded hash functions

One of the research questions, stated in chapter 1, was if it is possible to use a single hash function k times, seeding the input URL with a different salt each time, instead of using k different hash functions.

The test results in chapter 5 shows that there is no significant differences in using a seeded hash function k, instead of k hash functions. The results achieved for the false positive tests indicated that it is very feasible using a seeded hash functions instead of different hash functions, hence the temporal and spatial strategy tests was conducted using Jenkins hash with seeded values, instead of k different hash functions.

The advantage of using a seeded single hash function is that only one hash function is required. Since one hash function is required the best suitable hash function can be selected. This also makes the Bloom filter more predictable in insertion and query time, since it is only one hash functions that is calculating hash values.

The data tested in this thesis was URLs, hence this may not apply to all types of data. E.g. seeding names or dates may yield other results.

6.3 Temporal strategy

The goal of the temporal strategy was to provoke false positives on other elements upon a revisit, in a web crawler system. This is a very suitable strategy for web crawler that continuously revisit a web page to check for updates or changes.

The test in section 5.7 indicates that it is possible to use a temporal strategy to reduce the probability of a false positive. With a configuration of 1 Bloom filter with 3 hash functions and a bit-size multiplier of 1.7, the probability goes from 8.7927% and is reduced to 0.7687%. This is a significant improvement in the false positive probability. The other configurations for the filter shows that similar results is obtain, when using the temporal strategy.

In table 5.16 the configuration with 7 hash functions and 4.0 bit-size multiplier achieved a false positive probability of 0.000001% while the preceding configurations yields a false positive probability of 0%. This is because no matter how many hash functions is added, or how much the bit-size multiplier is increased, there is always a probability of false positive. This is because hash functions is used to determine the position to set in the bit array, and there will always be a probability that the hash functions collide, or that two different hash functions, or different seeds, hashes an element to the same value.

The temporal strategy have the advantages that is not a major modification of the Bloom filter, the only required change is that the hash functions, or seed values, is swapped when the crawler do a revisit of the web page. Since the temporal strategy is no major modification, existing systems may have the benefit of the strategy without any significant modification.

The results in the previous chapter also indicates that the temporal strategy is a feasible strategy to use in a web crawler system, or other similar system, where such a revisit happens.

6.4 Spatial strategy

We tested out our spatial strategy to see if it is possible to combined multiple Bloom filters to reduce the number of false positives. The goal of the spatial strategy was that a false positive in one filter, would not be a false positive in another filter. And thus discard false positives from one filter.

If we compare the result of one filter with 3 hash functions with two filter with 3 hash functions, in section 5.8, we see that the probability of a false positive drops significantly. This indicates that the spatial strategy is reducing the false positive probability.

However 2 Bloom filters with 3 hash functions yields a total of 6 hash functions. If we compare 2 Bloom filters with 3 hash functions with a single Bloom filter with 6 hash functions we see that this configurations yields similar results. From this we can deduce that the spatial solution is reducing the false positive probability, but a single filter with the same total amount of hash functions will provide approximately equal false positive probability.

If the filters in some way would be distributed, e.g. running on two different machines, the spatial strategy could be utilized to consume less resources on each machine. By dividing a single filter with 6 hash functions to two Bloom filter with 3 hash functions each, the Bloom filters could be distributed to multiple computers. The advantage of such a scheme is that it would require less memory on each computer. If such a strategy is introduced there must be some decision mechanism to conduct the queries on both filters and do a decision on if the element is contained or not.

6.5 Spatio-temporal strategy

On of the research questions, stated in chapter 1, was if it was possible to combine the two strategies to further reduce the false positive probability.

The test results in chapter 5 shows that it is possible to combine the two strategies to reduce the false positive probability. If we compare the results in table 5.20 with the results in table 5.16, if we use the spatio-temporal strategy and configure two filters with a bit-size multiplier of 1.7 and 3 hash functions, the spatio-temporal strategy yields a false positive probability of 0.0059%. If we compare this results to the temporal results, in table 5.16, we see that the false positive has dropped from 0.7687%.

It is worth to notice that 2 Bloom filters with 3 hash functions each, is in effect 6 hash functions total, hence if we compare it with a single filter we must compare it with a single filter that has 6 hash functions. If the previous result is compared to a single Bloom filter using the temporal strategy with a bit-size multiplier of 1.7 and 6 hash functions we get 0.0060%. If we do this comparison there is no significant reduction in the false positive probability. The other test results in table 5.20 and 5.16 shows similar results.

Chapter 7

Conclusion and further work

In this chapter we conclude this thesis, and propose further work and modifications.

7.1 Conclusion

In this thesis we have investigated methods to reduce the probability of a false positive in a Bloom filter, when the Bloom filter is used as a URL cache in a web crawler environment. We have tested two different strategies and combined them to a third strategy, which we also tested. The goal of this thesis was to propose novel spatio-temporal strategies to reduce the false positive rate introduced by the Bloom filter, in a URL cache.

In this thesis we have investigated three strategies, a spatial strategy which utilized multiple Bloom filter, and a temporal strategy which reduces the false positive probability over time, in a web crawler system and a combination of the latter two strategies. The intention of combination of the spatial and the temporal strategies, was to further reduce the false positive probability in the Bloom filter.

The test results for the spatial strategy shows that it is able to reduce the false positive probability. However, if the spatial strategy is compared to a single Bloom filter with the same total amount of hash functions, there is little or no gain in the false positive probability.

The temporal strategy yields very positive results. By swapping the hash functions upon a revisit the strategy is able to reduced the false positive probability significant. This strategy is also a very applicable strategy considering the small modification it requires to existing systems.

The two strategies were combined to see if this would further reduce the false positive probability. The test results indicates that it is able to reduce the false positive probability. However the same restriction applies with this strategy as the spatial strategy. If the strategy is compared to the same total amount of hash functions, there is little or no reduction in the false positive probability, and the results is only influenced by the temporal strategy.

The temporal strategy will be useful for URL caches in web crawler systems that continuously revisits web pages. The strategy is also feasible to use in other systems that has the same behaviour and needs to reduce memory consumption.

The temporal strategy is also very simple, hence it yields not large modification on a current system. This is an advantage for existing systems that utilizes Bloom filters and have similar behaviour.

7.2 Further work

7.2.1 Unknown number of URLs

A web crawler environment can be very unpredictable. The problem that affects each scheme, using a standard Bloom filter, is that the number of URLs that will be stored in the filter must be known before creating the filter, to ensure that the false positive probability does not grow beyond what is acceptable. In an unpredictable environment such as the web, this is a limitation, since the page must, in a worst case, be crawled to count the number of URLs before the page is actually crawler.

To overcome this limitation our temporal scheme should be tested with a Scalable Bloom filter, explained in [2], to see if the same reduction in false positive probability applies. The Scalable Bloom filter is then able to scale according to the number of URLs discovered on the crawled page.

Another approach to the problem would be to oversize the Bit array the first visit, and then count the number of URLs in the URL cache when the crawler is done on the page, and then store this meta-data in the storage. Upon a revisit the web crawler will know the number of URLs that the web page had on the last crawl, and can then set the size of the Bit array accordingly.

The drawback of this approach is that if the filter is to large the first visit it will use a lot of memory that is not utilized. If the Bit array is to small, the probability of a false positive will grow.

7.2.2 Finding the best fit hash function

False positives occur in a Bloom filter due to the fact that the Bloom filter utilize hash functions which are prone to get collisions, as described in section 3.4.

An approach to minimize the collisions would be to apply genetic programming, or similar, to generate close to so-called perfect hash functions. A perfect hash function is a hash function which is generated based on a known set of input data, and specialized for that data set. The perfect hash function does not yield any collisions, something that is almost impossible to accomplish in a practical setting like a web crawler system.

The idea is based on a combination of focused and continuous web crawling, as explained in section 2.3, where a web crawler at first crawls limited area of the web, and collects all URLs for that given area.

Then utilize genetic programming to generate a perfect hash function based on all the collected URLs. Since vast amount of new web content is generated daily, the generated hash function will not be able to function as a perfect hash function, after new content is added to the limited area to crawl. But the URLs will most likely utilize the same structure and be quite similar to the to the set of URLs the genetic programming algorithm generated the hash function from. Hence, the hash function would, presumably, only yield a number of collisions close to zero.

Appendix A

Appendix

A.1 Test results

A.1.1 Incremental spatial results

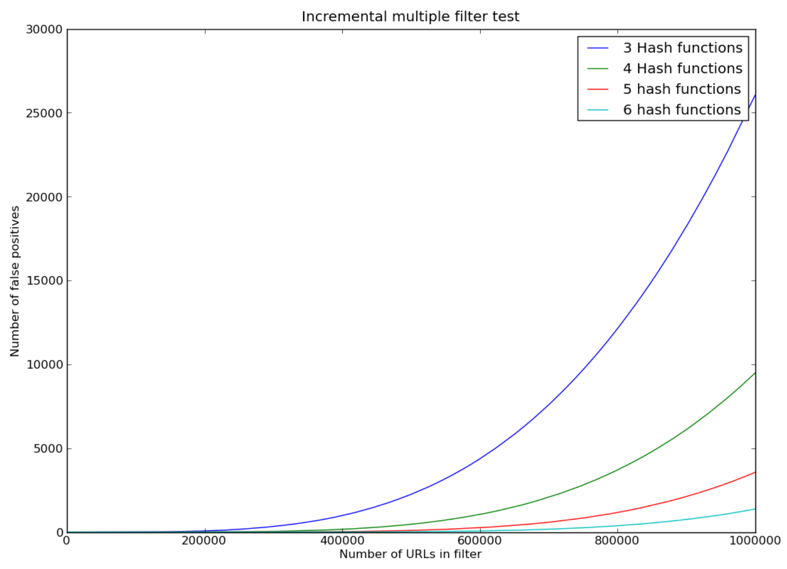

Figure A.1: 1 Bloom filter, incremental 1.7 bit-size

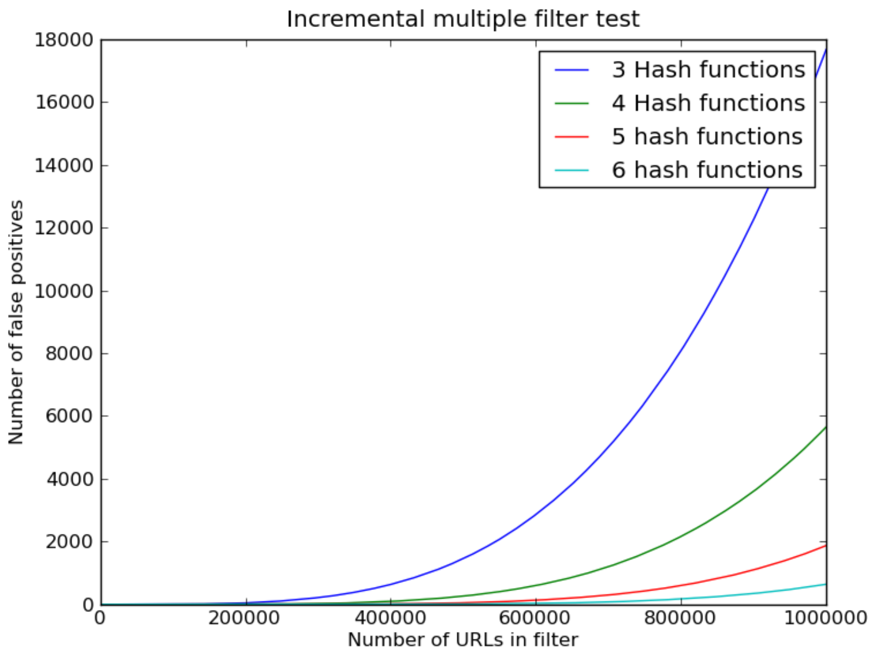

Figure A.2: 1 Bloom filter, incremental 2.0 bit-size

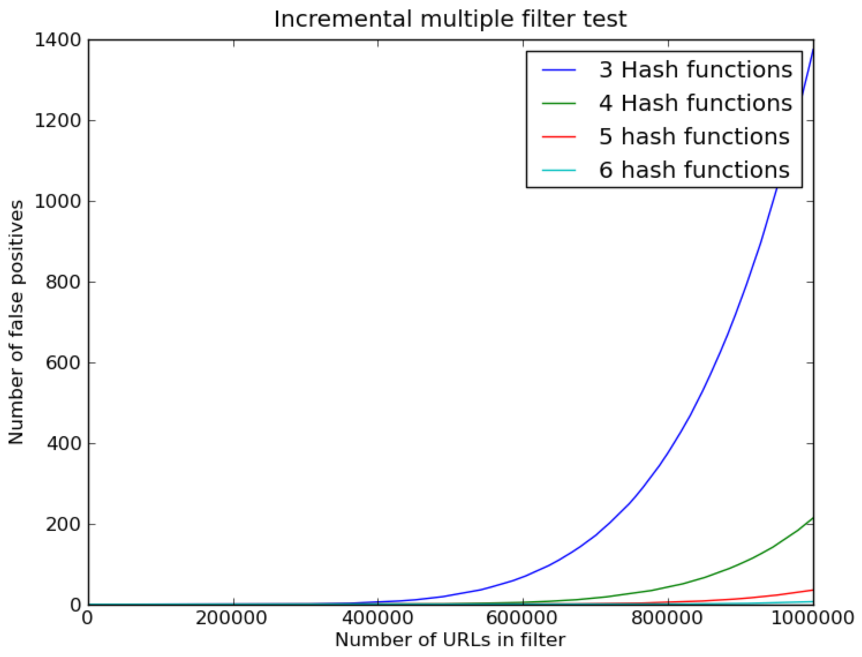

Figure A.3: 2 Bloom filter, incremental 1.7 bit-size

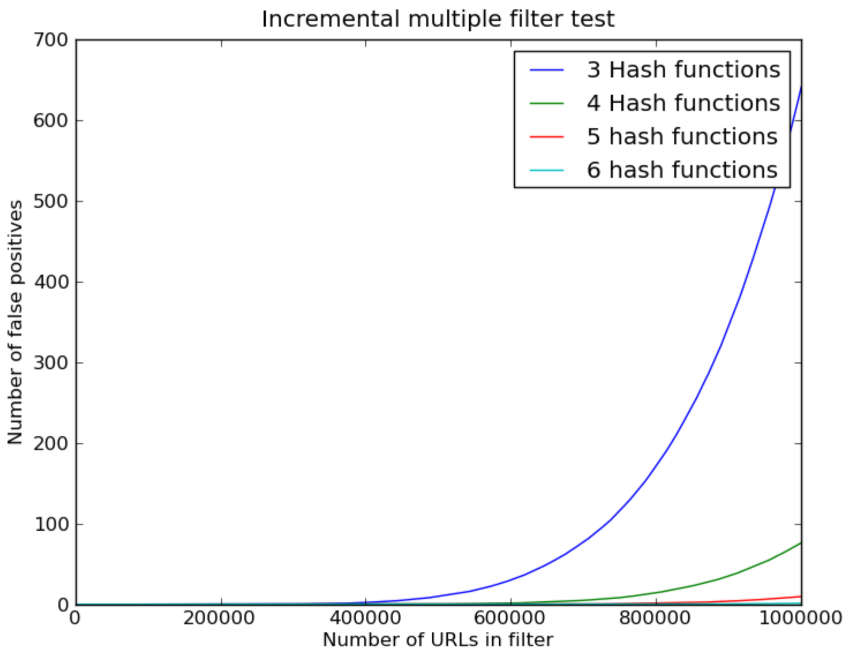

Figure A.4: 2 Bloom filter, incremental 2.0 bit-size

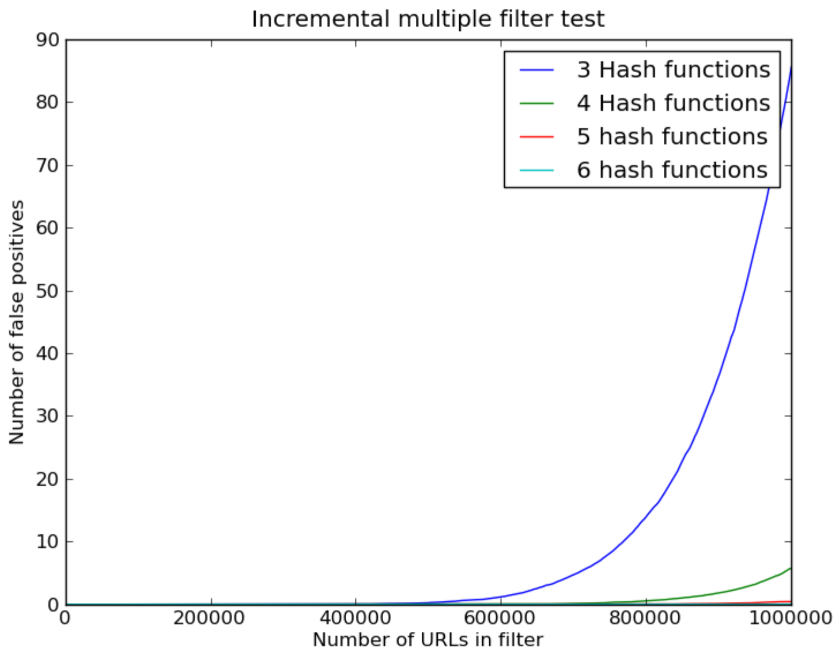

Figure A.5: 3 Bloom filter, incremental 1.7 bit-size

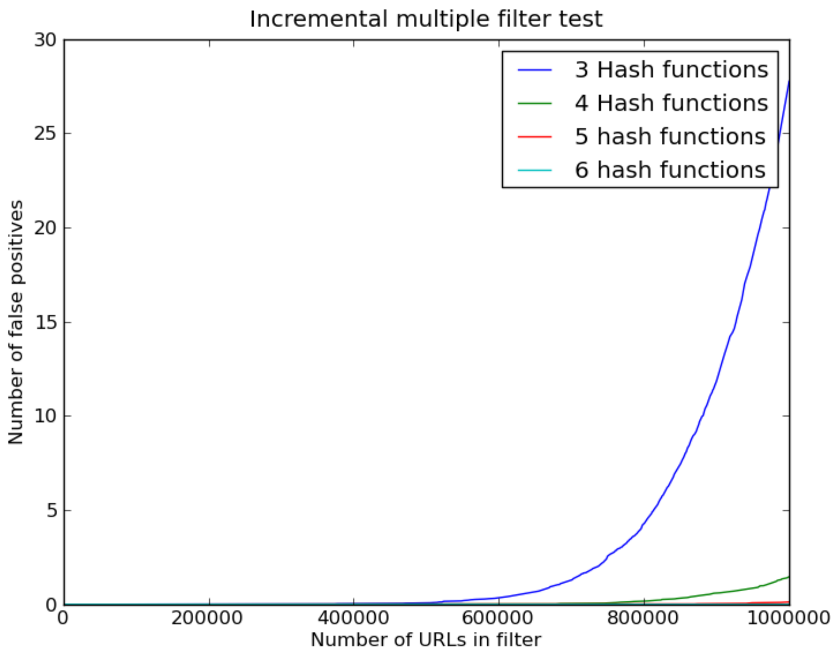

Figure A.6: 3 Bloom filter, incremental 2.0 bit-size

A.1.2 Query test

	1 Bloom filter					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	3296	3213	3297	3326	3347	3298
4 HF	4246	4266	4332	4386	4278	4201
5 HF	5287	5342	5403	5179	5311	5304
6 HF	6186	6275	6267	6309	6348	6282
7 HF	7270	7356	7487	7250	7270	7185
8 HF	8245	8301	8181	8101	8266	8214
9 HF	9215	9286	8966	9143	9064	9230
10 HF	10073	10064	10284	10046	9959	10135

Table A.1: Average query time for 1 filter in milliseconds

	2 Bloom filters					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	6324	6176	6355	6383	6422	6407
4 HF	8181	8318	8276	8301	8449	8466
5 HF	10096	10124	10151	10415	10174	10314
6 HF	12181	12039	11949	12315	11938	12279
7 HF	13883	14091	13792	13962	14007	14262
8 HF	15844	16184	15658	15865	16079	16109
9 HF	18096	17671	17616	17692	17709	17971
10 HF	19826	19442	19789	19629	20536	20146

Table A.2: Average query time for 2 filters in milliseconds

	3 Bloom filters					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	9342	9277	9634	9486	9430	9279
4 HF	12231	12186	12408	12272	11935	12285
5 HF	14971	15230	15015	15494	15151	15007
6 HF	17882	17962	17784	18264	17642	18220
7 HF	20503	20520	20500	20556	20639	21031
8 HF	23168	23754	23829	23412	23923	23626
9 HF	26625	26855	25930	26593	26839	27067
10 HF	29219	29783	29374	29996	28917	28801

Table A.3: Average query time for 3 filters in milliseconds

A.1.3 Add test

	1 Bloom filter					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	3354	3130	3390	3253	3279	3415
4 HF	4578	4297	4276	4403	4089	4352
5 HF	5263	5760	5541	4957	5856	6292
6 HF	6109	6872	6379	6384	6623	6332
7 HF	7000	7176	7413	7227	7209	7113
8 HF	8448	8872	8420	8650	8189	8323
9 HF	9395	9976	9258	9115	9102	9390
10 HF	10049	10485	10161	10544	10058	11018

Table A.4: Average add time for 1 filter in milliseconds

	2 Bloom filters					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	6565	6547	6297	6473	6259	6739
4 HF	8646	8866	8415	8446	8017	8412
5 HF	10868	10083	10014	9740	10750	10239
6 HF	12439	11778	13671	12124	12318	12099
7 HF	14473	14230	15173	14076	14570	15384
8 HF	16003	16311	16886	15589	17409	15990
9 HF	18456	18046	18465	19267	19209	17553
10 HF	19786	21259	20010	20183	21069	21298

Table A.5: Average add time for 2 filters in milliseconds

	3 Bloom filters					
	1.7 M	2.0 M	2.5 M	3.0 M	4.0 M	5.0 M
3 HF	9183	9908	9115	9514	9356	10139
4 HF	13326	12996	12812	12766	12418	12324
5 HF	16528	14823	15240	15934	15729	15288
6 HF	17593	17536	18044	18348	18365	18396
7 HF	21100	22748	20804	21824	23231	21261
8 HF	24032	23537	23976	26113	25796	26044
9 HF	28128	27329	28259	26458	26594	28458
10 HF	30273	29268	29585	29237	30443	31560

Table A.6: Average add time for 3 filters in milliseconds

A.2 Acronyms

ACM	Association for Computing Machinery
API	Application Programming Interface
CPU	Central Processing Unit
CFHF	Collision Free Hash Function
DNS	Domain Name System
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
MD5	Message-Digest algorithm 5
NIST	National Institute of Standards and Technology
OWHF	One-Way Hash Function
PRNG	Pseudo Random Number Generator
UiA	University of Agder
UML	Unified Modeling Language
URL	Uniform Resource Locator
WWW	World Wide Web

Bibliography

[1] *Computer Security, Principles and Practice*. Pearson Prentice Hall, 2008.

[2] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, “Scalable bloom filters,” *Inf. Process. Lett.*, vol. 101, no. 6, pp. 255–261, 2007.

[3] A. Appleby. Murmurhash. [Online]. Available: http://sites.google.com/site/murmurhash/

[4] S. Bakhtiari, R. Safavi-naini, J. Pieprzyk, and C. Computer, “Cryptographic hash functions: A survey,” Tech. Rep., 1995.

[5] (2003, Nov) How much information 2003? - internet. Berkeley. [Online]. Available: http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/internet.htm

[6] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, pp. 422–426, 1970.

[7] D. Boswell, “Distributed high-performance web crawlers: A survey of the state of the art,” 2003.

[8] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, “Network applications of bloom filters: A survey,” in *Internet Mathematics*, 2002, pp. 636–646.

[9] C. Castillo, D. A. Moffat, and D. G. Navarro, “Effective web crawling,” Tech. Rep., 2004.

[10] J. Cho and H. Garcia-Molina, “Effective page refresh policies for web crawlers,” Stanford InfoLab, Technical Report 2003-44, December 2003. [Online]. Available: http://ilpubs.stanford.edu:8090/604/

[11] S. Cohen and Y. Matias, "Spectral bloom filters," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2003, pp. 241–252.

[12] P. C. Dillinger and P. Manolios, "Fast and accurate bitstate verification for spin," in *In Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN*. Springer-Verlag, 2004, pp. 57–75.

[13] I. R. D. A. Gordon Mohr, Michael Stack and M. Kimpton, "An introduction to heritrix an open source archival quality web crawler."

[14] A. Heydon and M. Najork, "Mercator: A scalable, extensible web crawler," *World Wide Web Conference*, vol. 2, no. 4, pp. 219–229, April 1999. [Online]. Available: citeseer.nj.nec.com/heydon99mercator.html

[15] Interface instrumentation. [Online]. Available: http://java.sun.com/javase/6/docs/api/java/lang/instrument/Instrumentation.html

[16] B. Jenkins, "Algorithm alley: Hash functions," *Dr. Dobb's Journal*, 1997.

[17] C. Jing, "Application and research on weighted bloom filter and bloom filter in web cache," *Web Mining and Web-based Application, Pacific-Asia Conference on*, vol. 0, pp. 187–191, 2009.

[18] M. Kobayashi and K. Takeda, "Information retrieval on the web," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 144–173, 2000.

[19] D. P. Mehta and S. Sahni, *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004.

[20] B. Mozafari and M. Savoji, "A new collision resistant hash function based on optimum dimensionality reduction using walsh-hadamard transform," dec. 2006, pp. 149 –154.

[21] "The md5 message-digest algorithm." [Online]. Available: http://tools.ietf.org/html/rfc1321

[22] "Us secure hash algorithm 1 (sha1)." [Online]. Available: http://tools.ietf.org/html/rfc3174

[23] X. Wang and H. Yu, "How to break md5 and other hash functions," in *In EUROCRYPT*. Springer-Verlag, 2005.

[24] F. Weigel, “A survey of indexing techniques for semistructured documents,” Project thesis, Institute of Computer Science, LMU, Munich, 2002. [Online]. Available: http://www.pms.ifi.lmu.de/publikationen/#PA_Felix.Weigel

[25] M. Zhong, P. Lu, K. Shen, and J. Seiferas, “Optimizing data popularity conscious bloom filters,” in *PODC ’08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2008, pp. 355–364.

[26] J. Zobel, A. Moffat, and R. Sacks-davis, “An efficient indexing technique for full-text database systems,” in *In Proceedings of 18th International Conference on Very Large Databases*, 1992, pp. 352–362.