Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

ACTIVE CACHING FOR RECOMMENDER SYSTEMS

by
Muhammad Umar Qasim

Web users are often overwhelmed by the amount of information available while carrying out browsing and searching tasks. Recommender systems substantially reduce the information overload by suggesting a list of similar documents that users might find interesting. However, generating these ranked lists requires an enormous amount of resources that often results in access latency. Caching frequently accessed data has been a useful technique for reducing stress on limited resources and improving response time. Traditional passive caching techniques, where the focus is on answering queries based on temporal locality or popularity, achieve a very limited performance gain. In this dissertation, we are proposing an 'active caching' technique for recommender systems as an extension of the caching model. In this approach estimation is used to generate an answer for queries whose results are not explicitly cached, where the estimation makes use of the partial order lists cached for related queries. By answering non-cached queries along with cached queries, the active caching system acts as a form of query processor and offers substantial improvement over traditional caching methodologies. Test results for several data sets and recommendation techniques show substantial improvement in the cache hit rate, byte hit rate and CPU costs, while achieving reasonable recall rates. To ameliorate the performance of proposed active caching solution, a shared neighbor similarity measure is introduced which improves the recall rates by eliminating the dependence on monotinicity in the partial order lists. Finally, a greedy balancing cache selection policy is also proposed to select most appropriate data objects for the cache that help to improve the cache hit rate and recall further.

ACTIVE CACHING FOR RECOMMENDER SYSTEMS

by
Muhammad Umar Qasim

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Information Systems

Department of Information Systems, NJIT

May 2011

APPROVAL PAGE

ACTIVE CACHING FOR RECOMMENDER SYSTEMS

Muhammad Umar Qasim

Dr. Vincent Oria, Dissertation Co-Advisor — Date
Associate Professor of Computer Science, NJIT

Dr. Yi-Fang Brook Wu, Dissertation Co-Advisor — Date
Associate Professor of Information Systems, NJIT

Dr. Min Song, Committee Member — Date
Assistant Professor of Information Systems, NJIT

Dr. Dimitri Theodoratos, Committee Member — Date
Associate Professor of Computer Science, NJIT

Dr. Il Im, Committee Member — Date
Associate Professor of Information Systems, Yonsei University, South Korea

BIOGRAPHICAL SKETCH

Author: Muhammad Umar Qasim

Degree: Doctor of Philosophy

Date: May 2011

Undergraduate and Graduate Education:

- Doctor of Philosophy in Information Systems,
  New Jersey Institute of Technology, Newark, NJ, 2011
- Master of Science in Information Systems,
  New Jersey Institute of Technology, Newark, NJ, 2005
- Master of Business Administration,
  Hamdard University, Karachi, Pakistan, 1999
- Bachelor of Computer Science,
  Hamdard University, Karachi, Pakistan, 1997

Major: Information Systems

Presentations and Publications:

Hyuck Han, Michael Houle, Vincent Oria and Umar Qasim, "Caching without Replacement for Approximate Similarity Search," *to be submitted to PVLDB.*

Umar Qasim, Vincent Oria, Michael Houle, Yi-Fang Wu and M. Tamer zsu, "An Active Cache for Recommender Systems Based on Partial-Order," *submitted to ACM Transactions on the Web.*

Michael Houle, Vincent Oria and Umar Qasim, "Active Caching for Similarity Queries Based on Shared-Neighbor Information," *Proceeding of the 19th International Conference on Information and Knowledge Management,* Toronto, Canada, October 2010.

Umar Qasim, Vincent Oria, Yi-Fang Wu, Michael Houle and M. Tamer zsu, "A Partial Order Based Active Cache for Recommender Systems," *Proceeding of the 3rd ACM Conference on Recommender Systems,* New York, USA, October 2009.

I would like to dedicate this doctoral dissertation to my father, Muhammad Qasim, without his continuous encouragement I could not have completed this process and to my wife, Munazza, for being a candle flame in dark times and a patient endurer.

ACKNOWLEDGMENT

With a deep sense of appreciation, I would like to express my sincere thanks to Dr. Vincent Oria for his immense help in planning and executing this work. His support, encouragement and reassurance are greatly acknowledged. Heartfelt thanks to Dr. Yi-Fang Brook Wu for her valuable suggestions and guidance which helped me in completing this work.

My sincere thanks to Dr. Min Song, Dr. Dimitri Theodoratos and Dr. Il Im for reviewing this work and providing me important suggestions.

I would like to acknowledge Dr. Michael E. Houle in providing great help and guidance during this research. Special thanks are due to Dr. Michael Bieber for his enormous support during difficult times.

I would like to express profound gratitude to my family and friends for uncomplaining patience, perennial support, and enormous encouragement during my studies. I wish I would never forget the company I had from my friends in New Jersey. I am thankful to Mumtaz, Sajjad, Umar, Akhtar and Nadeem for their countless cooperation, help and encouragement.

Partial support for this research was provided by the National Science Foundation under grant DUE-0434998. I am very thankful to the National Science Foundation and the Information Systems department for the financial support.

TABLE OF CONTENTS

Table of Contents
(Continued)

Chapter Page

Table of Contents
(Continued)

Chapter Page

Table of Contents
(Continued)

Chapter Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

List of Figures
(Continued)

Figure **Page**

List of Figures
(Continued)

Figure **Page**

List of Figures
(Continued)

Figure **Page**

List of Figures
(Continued)

Figure **Page**

List of Figures
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Introduction

With the increasing popularity of the World Wide Web, amount of information and number of users are growing exponentially. However, this increase, particularly in database backed web sites, has created challenges for web developers to provide efficient solutions. Traditional applications such as file transfer, news and email need more throughput but can tolerate delays. However, applications of interactive nature require latencies on the order of seconds [15]. Recommender Systems, being computationally intensive and interactive applications, cannot tolerate access latency. Although advanced more powerful physical resources can help to improve the performance yet there is dire need for further improvement using optimization techniques.

In many databases and Web applications, caching is often employed to improve response time and reduce the server workload. A cache is a temporary storage area where data can be stored for quick access. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or recomputing the original data, so that the average access time is shorter. Caching can improve the performance of an application by reducing access latency, server load and network traffic. Caching strategies can be divided into two broad classes: traditional 'passive' caching, and the more recent 'active' caching. With passive caching, where the server query result is retrieved either directly from the cache or from the disk, the effectiveness of the operation is guaranteed. Passive cache management strategies generally seek to fill the cache with result lists for the most popular queries, and utilize effective replacement strategies to maximize the overall performance. In general, only limited performance gains are possible with passive caching. Active caching techniques attempt to improve upon the performance of passive caching by

synthesizing a query result from stored information whenever the sought-after result is not explicitly present. This form of caching is referred to as 'active' since the cache can be considered to function in a limited query-processing role [83].

The active caching techniques proposed in the research literature to date are extensions of relational database caching strategies that aim to answer Boolean queries: the caching strategies attempt to make use of the stored results of past queries to generate a new result that satisfies the containment criteria of the current query. For recommender system queries returning a ranked list of the top-k relevant objects this form of active caching cannot be applied. This dissertation proposes an active caching strategy specifically designed for recommender system queries or top-k similarity queries (also known as k-nearest-neighbor, or k-NN queries).

1.2 Background and Motivation

This research addresses the problem of access latency in recommender systems. In contrary to effectiveness studies dominated in recent recommender system research, this dissertation focuses on efficiency aspect of these applications. Recommender systems aid users in finding useful information according to their interests. However, delay in providing this aid can lead to user frustration and result in non-usage of such systems. Therefore efficiency is very crucial aspect for the success of these applications and must be addressed.

The specific objectives of the research are to address the latency problem in recommender systems and find a way to solve or at least mitigate the problems. Although performance optimization work has been done in the related domains i.e., search engines, yet none of the existing studies have specifically addressed recommender systems. These approaches improve the performance to a certain level and can be extended to work for recommender system. This dissertation focuses on investigating an optimization solution specifically designed for recommender systems which performs better than already available techniques.

In many databases and Web applications, caching is often employed to improve response time and reduce the server workload. Caching techniques have seen significant success in the query processing both in databases and web applications. Traditional caching approaches only attempt to answer those queries whose result is available in the cache. These approaches generally seek to fill the cache with result lists for popular queries, and to utilize effective replacement strategies to maximize overall performance. In general, only limited performance gains are possible with this type of caching. More recently, active caching techniques have been developed that use the results of prior queries to answer related queries. However, these caching strategies attempt to make use of query containment but this form of active caching cannot be used with recommender system queries.

The particular problem to be solved in this research is designing and implementing an active caching strategy that can improve upon the performance of traditional caching solutions for recommender systems. This approach is expected to work better than already available caching methods and should work for all types of recommender systems. Major research questions include; How to design an effective and efficient caching solution for recommender systems? How to design a more general and effective similarity measure for active caching? How to select the objects in the cache for a caching with no replacement?

1.3 Proposed Methodology

This research first reviews the related work in recommender systems and web-caching domains and then presents the active caching solution for recommender system. Recommender systems normally differ how the resultant list of recommendation are computed. Traditional caching solutions can easily be used with any type of recommender system. With traditional caching, where the server query result is retrieved either directly from the cache or from the disk, the effectiveness of the operation is guaranteed. Traditional cache management strategies generally seek to fill the cache with result lists for the most

popular queries, and to utilize effective replacement strategies to maximize the overall performance. In general, only limited performance gains are possible with this type of caching. Active caching technique attempt to improve upon the performance of tractional caching by synthesizing a query result from stored information whenever the sought-after result is not explicitly present. Thus if implemented, active caching can provide significant improvement in performance over traditional caching solutions.

This dissertation proposes an active caching solution for recommendation systems which also works with any other application that uses top-k similarity queries e.g. contextual advertising, image retrieval etc. The proposed active caching solution not only returns cached results, but also actively estimates answer for queries whose results are not present in the cache, by aggregating those results stored in the cache for related queries. This dissertation first introduces the basic structure of the proposed active caching solution. The proposed approach is capable of efficiently synthesizing answers for non-cached queries using the partial order lists available in the cache. It uses both cached lists of objects in the neighborhood of query objects, as well as inverted lists derived from these neighbor lists. The basic solution drives The basic active caching technique utilizes the partial order that can be used with any type of recommender system. The basic solution is built upon the monotonicity feature of the cached partial order lists and uses aggregate functions to assess the similarity between two objects. This solution successfully estimates the answer for non-cached queries. However, due to inherent dependency on monotonicity, result accuracy can be lower for datasets having lower levels of monotonicity.

To improve the accuracy of estimated results, a shared-neighbor similarity measure is introduced latter in the dissertation. This method assess the similarity between two objects in terms of the number of other objects in the common intersection of their neighborhoods. The proposed method is general in a sense that it does not require that the features be drawn from a metric space, nor does it require that the partial orders induced by the similarity

measure be monotonic. It helps to improve the accuracy of query results that are actively processed from the cache.

Finally, this work proposes a greedy balancing strategy for the selection of appropriate cache data in order to answer maximum number of queries.The proposed greedy balancing heuristic for the selection of the cache content provides a good coverage over the range of possible queries, and improves both the hit rate and average recall even for small cache sizes.

For the implementation of active caching approach, a two dimensional memory based data structure has been introduced. This data structure keeps lists of cached queries as forward lists whereas inverted lists for each object in the cache is maintained as well. Forward lists help in answering cached queries where as inverted lists along with the forward lists are used to estimate the answer for non-cached queries.

The evaluation focuses on the efficiency and effectiveness of the proposed methods and the optimal settings to achieve the best performance. Efficiency of the system is measured by hit rate, byte hit rate and execution cost. Hit rate is the proportion of requests that are answered by the cache over the total number of requests whereas execution cost is the total time to process given number of queries. Performance effectiveness is measured by recall i.e., the proportion of the result returned from cache that would also appear in a top-k query if requested from the recommender system.

1.4 Research Scope

Recommender System is a very broad area and several techniques are available to implement these systems. This study proposes active caching solution for any type of recommender system in general. Although this solution works with other applications that produce ranked lists as an output, however, this study focuses on recommender system domain. Evaluation is focused on two most famous techniques e.g. content based and collaborative filtering to test the proposed approach due to the unavailability of datasets for other types of systems.

There are several variations in the implementation of CB and CF systems. Mainly these variations are due to the type of method used to compute relevancy amongst the objects in the collection. The proposed solutions work for any variation of these recommendation techniques. The proposed active caching approach is targeted to provide solution for any type of recommender system however, it can also work for any nearest neighbor application.

Recommender systems use various methods to compute relevancy scores amongst the objects in its collection. While integrating these recommender systems in other applications, the access to this internal relevancy information might be possible in some cases and not possible in other cases. The proposed approach does not make use of actual relevancy scores therefore can be used in both situations. One limitation of the partial order based approach is that it produces lower recall rates with the datasets having lower levels of monotonicity in the data. Shared neighbor approach overcome this limitation because it does not depend on monotonicity in the underline data. The current study does not consider dynamic changes in the object collection and the relevancy scores are computed for the fixed collection. One extension to the study could be taking into consideration dynamic changes in the object collection.

Main focus of this study is to provide active caching architecture along with appropriate cache management policy for the selection of objects to be kept in the cache. This study uses proactive cache loading approach where cache is loaded upfront and no cache replacement policy is used. However, due to the unavailability of recommender system query logs, comparison with other cache replacement policies is not done.

The evaluation focuses on the performance of the proposed methodology, which is measured by the cache hit rate, execution cost and the average recall of the results processed from the active cache. Other factors, such as system usability and user's satisfaction, though might be important to know, are not explored in this study.

1.5 Intellectual Merit and Contributions

Most of the current studies in the area of recommender system have focused on the effectiveness of generated recommendations. However, performance of these systems is at risk if efficiency is not addressed properly. Caching has been successfully used in many applications to improve response time and reduce the server workload. Traditional caching approaches can easily be used with recommender systems however, limited performance gains are possible with already available caching. This study proposes an active caching solution for recommender systems that attempts to improve upon the performance of traditional caching by synthesizing a query result from stored information whenever the sought-after result is not explicitly present.

This study has targeted to address the issue of efficiency in recommender systems and provides significant contributions in various ways. First, the conventional approach is to fill the cache with those items most likely to be requested in future queries, the partial order based active caching solution can instead support a form of *data interpolation*, in which the cache is used to actively process most if not all query results. Second, it proposes the design of shared-neighbor similarity measure for active caching to make the active solution more general and effective. It allows for variation of such parameters as the size of the cache, the length of ranked lists stored in the cache, and the number of items requested by the query. Third, the greedy balancing cache selection strategy balances the size of the inverted cache lists through reduction in variance of the lengths of these lists, thereby balancing the frequency of appearance of objects in the cached top-k neighbor lists. By achieving a better inverted list balance, it provides a better uniform coverage of the query range, and increases the spatial locality from which most if not all query results can be actively generated. The work proposed in this study is very general and can be used with any nearest neighbor application with top-k similarity queries. Also it uses techniques to drive ranking functions without any knowledge of the similarity values used in producing

these lists and does not require that the features be drawn from a metric space. Three main contributions of this work are

Partial Order Based Active Caching Approach: Chapter 4 proposes a partial order based active caching approach. The main contribution of the partial order based active caching algorithm is that it effectively estimates the answers for non-cached queries. The algorithm builds upon monotonicity amongst partial order lists and aggregate functions help to estimate the most relevant objects. The objective is to serve as many requests from the cache as possible, by implementing a policy that takes advantage of the freedom to provide a similar item. Furthermore, the solution is robust, unlike traditional approaches it can even work in cases where queries are less likely to be repeated and can be applied even when non-metric and probabilistic approaches are used to produce query results [105].

Shared Neighbor Similarity Measure: Chapter 5 proposes a shared-neighbor similarity measure which makes the active caching solution more general and improves the accuracy of estimated results by eliminating the dependency on monotonicity. It introduces a general model, the Cache-Estimated Significance (CES), for the estimation of the results of top-k similarity queries using shared-neighbor similarity measure on cached information. The model does not assume any knowledge of the methods or similarity measures used, nor does it require that the partial orders induced by the similarity measure be monotonic and as such can be applied even when non-metric and probabilistic approaches are used to produce query results. The main contribution of the CES approach is to facilitate the design of shared-neighbor ranking formulae for active caching that allow for variation of (and comparison across) such parameters as the size of the cache, the length of ranked lists stored in the cache, and the number of items requested by the query . The CES model improves upon the performance of partial order approach by eliminating the dependency on monotonicity and helps to achieve higher recall for queries whose answers are estimated from the cache [49].

Greedy Balancing Cache Selection Policy: Chapter 6 proposes a greedy balancing strategy, CES-GB, for the selection of appropriate cache data in order to answer the largest possible number of queries. The proposed active caching strategy has been shown to depend on the frequency in which the query object appears together with result objects in the lists stored in the cache. The main contribution of the CES-GB algorithm is that it balances the size of the inverted cache lists through reduction in variance of the lengths of these lists, thereby balancing the frequency of appearance of objects in the cached top-k neighbor lists. By achieving a better inverted list balance, CES-GB provides a better uniform coverage of the query range, and increases the spatial locality from which most if not all query results can be actively generated. CES-GB provides significant improvement in the hit rate and average recall for small caches. Since the size of cache memory is usually much smaller than the total dataset size, this approach can have a great practical impact. Even for small caches, CES-GB may be sufficient to answer all queries actively, without ever referring to the original dataset. This form of active caching therefore has the potential to serve as a scalability technique. With the explosive growth of data repositories and the popularity of similarity-based applications, the CES-GB approach opens doors for new forms of indices based on data sampling [45].

This research is not only going to contribute to the recommender systems domain, but also provide guidelines to develop caching solutions in many other areas. For some applications, it may even suffice to answer all similarity queries actively, without ever referring to the original data. Active caching could thus serve as a scalability technique, as it provides the basis of space- and time-efficient approximation of large databases.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 provide review of literature related to the study. It presents the background of recommender systems

and overview of important recommender system techniques. It also discusses available caching techniques and various strategies to implement them. Chapter 3 discusses possible approaches for implementing caching solutions for recommender systems. It also outlines the research questions, provides information about the datasets and performance measures that will be used to assess the performance of proposed solution. Chapter 4 describes the proposed framework in detail for the partial order based active caching solution. The methodologies as well as the algorithms and test results are presented. Chapter 5 proposes a shared-neighbor similarity measure for active caching solution that helps to improve the accuracy of results, including the architecture, algorithm for estimation and experimental results for this approach and comparison with the basic partial order approach. Chapter 6 proposes a greedy balancing cache selection policy which provides a good coverage over the range of possible queries, and improves both the hit rate and average recall for small cache sizes. The dissertation is concluded with the expected contributions of this research in Chapter 7.

CHAPTER 2

LITERATURE REVIEW

This chapter provides background information of recommender systems, goals of recommender systems and various techniques used for building recommender systems. It also provides insight into caching techniques, ways to implement these techniques and discuss possibilities of implementing caching solution for recommender system.

2.1 Recommender Systems

Information available on the World Wide Web has been growing enormously. Web users are often overwhelmed by the amount of information available creating an information overload problem. Information overload is becoming more and more complex with the rapid growth of web for the information seekers. It is difficult to make choices among the alternatives without enough personal experience and information overload makes it even more difficult. Many techniques have emerged to assist the web users in finding the desired information more efficiently and effectively. One of the emerging techniques is recommendation system which assists users in finding desired information. In everyday life, people rely on "word-of-mouth" recommendations to make decisions [97]. Recommender system automate this natural social process to assist users. These systems act as personalized decision guides for users, aiding them in decision making about matters related to personal taste. Recommender systems attempt to reduce information overload by providing a subset of items from a universal set that are likely of interest to the user. In its most common formulation, the recommendation problem is reduced to the problem of estimating likelihood of the items that have not been seen by a user. This is an important application area and the focus of considerable recent academic and commercial interest.

Recommender system roots can be traced back to the cognitive science, information retrieval, forecasting theories, approximation theory, and management science [97]. Term recommender system was first introduced by Resnick and Varian as a system which accepts user input, aggregates them, and returns recommendations to users [108]. Research in this field started after a series of shifts in information systems research. In the 1970's great deal of IS research was focused on information retrieval systems [114]. The emphasis of such research was on retrieving information deemed relevant to queries. Development of vector space model was the result of one of those research studies which permitted similarity to be measured by the cosine of the angle between vectors [115].

In the 1980's, with the rapid increase in the amount of electronic information, researchers began to focus on removing irrelevant information rather than retrieving relevant information which ignited research in the area of information filtering [102]. Belkin and croft in one of the early research papers explained the idea of information filtering as a process which involves removing persistent and irrelevant information over a long period of time [13]. Information filtering later was termed as content-based filtering to the recommender system community and has since been used in many domains [7] [95]. Content-based systems model content features of objects and recommend items by querying such features against preferences of the user [67]. Selective dissemination of Information was one of the first information filtering systems [51]. This system provided information about the availability of resources meeting the user's search parameters. The selection was based on a user profile which has a list of keywords that described their interests. Since then content-based systems have be used in many domains; however, it is most effective in text-intensive domains, e.g. digital libraries, which account for only a portion of the artifact landscape. This limitation led the researchers to implement work on alternate information filtering solutions.

One of the major advancement in the area of information filtering was the initiation of a filtering system by Goldberg et al. Their system Tapestry became the first known

recommender system which was based on collaborative filtering technique [41]. Tapestry mail system developed at the Xerox used user reactions to the documents they read. It then used these reactions to filter incoming streams of electronic documents. In this way these systems result in filtering items for a user that similar users filtered. Collaborative filtering introduced a major shift in information filtering research and since been applied in many publicly available systems, and even some commercially available systems.

GroupLens is one of the first known project in the recommender system domain and the main goal of this project was to explore automated collaborative filtering [63]. Soon after, collaborative filtering technique was applied in filtering the information in Usenet news [94]. Ringo agent was one of the first applications that provided personalized music recommendations, which became available on July 1st 1994 [124]. In this method the users provide the ratings of the music articles. Based on these opinions the user profile, which changes over time, is created. The profile enables to create the recommendations by utilizing the social filtering method. This method can be treated as the automation of the wordofmouth recommendation [124]. The application that utilized concept of the Ringo system was Fireflys system. This technology was further developed by Yahoo and Barnesandnoble who signed up to use it [58]. One of the most famous implementations of collaborative filtering is done by the book dealer Amazon.com that introduced the BookMatcher system. At the beginning the BookMatcher was used for book recommendations, but later on, the system started to recommend other types of items, using also other methods of recommendation [58].

In recent years, online recommender systems are successfully providing assistance to the users. Recommender systems are being successfully used by many online businesses like amazon, ebay etc. E-commerce sites use these systems to suggest items to the customers which assist them to determine which products to purchase. These items can be recommended based on the most selling items on a site, on the demographics of the customer, or through analysis of the past buying behavior of the customer to predict his/her future

buying behavior. Recommendations could be provided by suggesting products to the customer, providing personalized product information, summarizing community opinion, and showing community critiques. Recommender systems help E-Commerce sites in improving sales by helping customers find products they want to purchase; converting browsers into buyers; improving by recommending additional products for the customer to purchase; improving loyalty by creating a relationship with the customer [121].

Much of the recent work in recommender systems is focused on e-commerce applications. However, many other domains have also opted to use this technology. iTunes use it to recommend top songs, Bloglines suggest users about the similar blogs, the NYTimes guides people to show most emailed articles, Del.icio.us uses this technology to recommend most popular bookmarks, Netflix & Reel use it to recommend movies to their users. Content-based book recommender system developed by Mooney & Roy uses information extraction and a machine-learning techniques to recommend books [95]. Recommender systems for web pages [101] and newsgroup messages [70] have also been developed to provide recommendations in these domains. The list is growing rapidly and it seems like this technology will become an essential part of most online applications [120].

2.1.1 Goals of Recommender Systems

Recommendation systems suggest objects and services that are likely of interest to the user. The aim is to help the potential user to select the appropriate object and hence, act as decision support systems. Furthermore these systems serve as a marketing tool for the ecommerce stores to attract customers. In short, the main objectives of these systems are to cop with the information overload problem, helps customers in decision making, and helps to increase sales for e-commerce businesses [97].

Information overload is a common problem in the digital era and recommender systems are a popular and effective choice for combating information overload. Information overload is believed to occur when the information received poses an obstacle rather than an

aid to the user [12]. This situation causes potentially useful and even critical information to be overlooked and may result in productivity losses. In situations where amount of information is overwhelming, the knowledge of which information is useful and valuable matters most because the chances of overlooking critical information are much higher [46]. Recommender systems are able to select a small subset of objects that seems to fit users' needs and preferences from a much bigger dataset. Users don't have to browse through all the objects in a dataset to find objects of interest thus effectively help to cop with the information overload problem [94] [4] [60].

Recommendation systems are applications that identify list of objects of potential interest to a user based on the user's interaction with a system. By restricting the number of suggested objects, recommender systems help people in decision making [97]. In general, these systems help customers in making decisions like what items to buy, which news to read next or which movie to watch, much faster than by the regular browsing. These systems provide a web-based decision support system that analyze the users skills, attitudes, preferences, etc., and then compute relevant information to support their decisions concerning actions on a particular website.

Recommender systems can also be used as a marketing tool as they can help to increase sales for e-commerce websites. Recommender system field is growing rapidly and adopted as business tools and changing the way people do business over the internet. Large business organizations adopting this tool to improve their sales [121]. Recommender systems help to increase sales by converting browsers into buyers, cross-sell and improving loyalty with a customer. Often site visitors pass browse through a website without purchasing anything. Recommender systems can suggest items of interest to visitors and help to converter them into buyers. These systems also help in increasing the average order size through cross-sell. For example, a user might be suggested additional items in the checkout process based on the products already in the shopping cart. Recommender systems can improve loyalty through building relationship between a seller and a buyer .

Sellers are eager to learn about the behavior of their users which can help to customize their interaction with the users [121].

2.1.2 Types of Recommender Systems

Recommendations suggested by a recommender system can be obtained in different ways [108] [120]which results in different implementations of these systems depending on the information and techniques used to compute recommendations. content based [101] [23], collaborative [101] [46], demographic, knowledge based [20], utility based, and several kinds of hybridations among these methods.

Usually recommender systems are categorized by their approach to compute list of recommendations. Recommender systems suggest documents, products, services etc. to the users using various methods. These methods differ by the type of a background data as well as the algorithm that is used to generate the recommendations. Adomavicius mentioned three main categories of recommender systems that are most popular and significant; collaborative filtering, content-based filtering, and hybrid methods [4]. Robin Burke distinguished five techniques of the recommendation systems: collaborative, content-based, demographic, utility-based, and knowledge-based [20]. Schafer et al. [121] listed six categories as most current recommendation systems: raw retrieval, manual selection, statistical summarization, attribute-based, item-to-item correlation, and user-to-user correlation. In a recent study Castellano and Martinez mentioned content based, collaborative , demographic, knowledge based, utility based and hybrid techniques as major recommender system approaches [23].

Based on the above mentioned classifications, recommender systems could of types; raw retrieval, manual selection, statistical summarization, attribute-based, content based, collaborative, knowledge based, demographic, utility based and hybrid systems. Raw retrieval or “null recommender” system provides customers with a search interface through which they can query a database of items and is technically not a recommender application

[120]. Similarly, in manual selection method recommendations are manually selected by editors, artists, critics, and other experts. This process does not use computer computation at all and "human recommenders" provide the recommendations therefore, not a pure recommender application.

Amongst the remaining types, few are variations of a similar recommendation technique and can be categorized under broader types. For example, statistical summarization technique provides recommendations by statistical summaries of the community opinion. This technique can be categorized as a type of collaborative filtering because it uses the rating / history information similar to other CF systems. This technique provides non-personalized recommendations, same recommendations for all the users, which are based on with in community popular items using rating/history data. Similarly, in demographic based systems, demographically similar users are identified and only rating / history from this set of users are used to compute recommendations. As such this is also a variant of collaborative filtering algorithm. Attribute based system uses properties of the items and users interest in those properties. Attribute based systems use customer's profiles that indicate likes or dislikes to process recommendations for a user [120]. This type of technique has also been used in some content-based systems where user history or profile information is used along with the content information to make recommendations. In utility based system features of items in the set are identified. Then a utility function that defines the user's preferences is applied to this set to find out the relevance of each item to this function. This type of system is also similar to content based systems that take into account user's preferences either implicitly or explicitly. Explicit information could be possible form profiles and implicit information can be extracted from user rating / history data.

Based on the above discussion, recommendation techniques can be categorized into four broader recommendation techniques; content based, collaborative filtering, knowledge based and hybrid as shown in the Figure 2.2 and explained in the Figure 2.1.

Recommendation Approach	Computation Process	Variations
Content-Based	Identify features of objects in set **O** and compute k nearest neighbors of object **o** in the set	• **Distance based** : Uses Euclidean or other distance functions to compute K-NN • **Similarity based** : Uses Cosine or other similarity functions to compute K-NN • **Clustering based** : Uses a clustering method to compute K-NN • **Attribute based:** Consider user **u**'s rating /history or preferences also along with the object features. • **Utility Based** – Uses a utility function defined by **u** over objects in **O**
Collaborative Filtering	Based on ratings /history data Identify users in the set **U** similar to **u** and predict objects for **u**	• **User to user correlation** : Find out similar users from rating / history data • **Item to item correlation** : Find out similar items from rating /history data • **Demographic based** - identify demographically similar users and only use rating/history from those. • **Statistical summaries**: Uses popularity measure / aggregate functions.
Knowledge-Based	Identify features of objects in **O** and how these matches with a user **u**'s profile	
Hybrid	Combine two or more of the above mentioned approaches.	

Figure 2.1 Table categorizing famous recommendation techniques discussed in the literature according to the implementation method.

Content Based: Content-based (CB) recommender systems recommend items based on the products the customers have expressed interest in. For example, if a customer has checked an item or placed an item in his/her shopping cart, the recommender system may recommend items similar to that. Generally in content-based approach objects are recommended based on correlation between objects in the collection, but in some systems, like attribute based, user profile is also managed and used. Content-based system uses the description of the items that were previously watched or purchased by the customer and/or evaluated by them in a positive way. Content-based system recommends items to the customers similar to the items they liked in the past [60] [94].

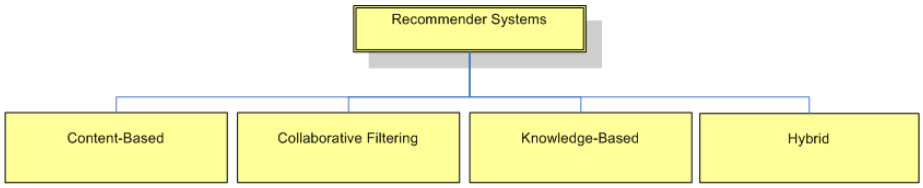

Figure 2.2 Classification of recommendation techniques into four broader categories.

Content-based systems are normally based on observations of the user's selection and generate recommendations automatically. Automatic system doesn't require any explicit input from the user whereas manual system requires the user to explicitly type in several items of interest in order to generate recommendations. Another variation in recommender systems is whether the history rating information is required or not. Transient systems do not need to know any history / rating information about the user to generate recommendations. On the contrary non-transient or persistent system requires history / rating information on the products he/she has selected or purchased in the past to produce recommendations for a particular user. Content based systems are usually transient, for example, moviefinder and reel.com systems recommend products to a user based on another product that user liked it in the past. These systems are transient as well as automatic, because they do not require any action or information about the customer. CDNOW application on the other hand is different. In this album advisor user has to type in a set of artists and system them provides recommendations based on this list. This application is still transient, however, is not totally automatic [120]. Some of the CB systems are also persistent and keep information for each user to generate recommendation. Output of a content based system is a list of items that are most similar to the one that customer has shown interest in. This list is ranked in the order of relevancy to the item that the user has shown interest in. This relevancy can be calculated by various measures for example, cosine similarity, Euclidean distance, Pearson's correlation etc.

The content-based approach to recommendation has its roots in the information retrieval (IR) community. In 1992, Belkin and Croft compared information filtering and

information retrieval systems in the ACM special issue on information retrieval. They explained that information retrieval systems return relevant information in response to a short-term information-seeking goal posed via queries, whereas information filtering involves removing persistent and irrelevant information over a long period of time [13]. Information filtering later became known as content-based filtering in the recommender system domain [7] [95]. Content-based systems model content features of objects and recommend items by querying such features against preferences of the user [67]. Selective dissemination of Information was one of the first information filtering systems [51]. This system provided information about the availability of resources meeting the user's search parameters. The selection was based on a user profile which has a list of keywords that described their interests.

Content-based systems have been used in many domains and particularly are very effective in text-intensive domains. Content features are more naturally available textual objects, such as books and articles. The content of a book or article available for recommendation generation could be title, abstract, authors, and even the full text. Most of the current content-based filtering approaches combine techniques from Information Retrieval and Machine Learning. Many approaches treat the recommendation problem as a classification problem using supervised learning techniques [11] [95]. Using these classification techniques the objects are categorized in predefined categories based on their content features. Some other approaches treat content-based filtering as a regression problem, in which a statistical model, especially a regression model, is learned from the training data and used to predict the ratings of documents unknown to a user [135]. Content-based filtering technique is usually based on three main modules: representations of user profiles, modeling documents and matching them to user profile representations.

User profile creation can be achieved through a manual process or through automatic categorization techniques. Manual approaches have been used to develop user's profiles in a book recommendation system [109]. In this system while evaluating recommended

books, users have to explicitly tell the system the information about themselves. Automated techniques provide new ways of exploiting content-based filtering to generate recommendations. Supervised learning approaches automatically form user profiles from a set of training documents [95]. In this process first a user selects and rates a few training books on a given scale. Classification system then builds a model for the user by analyzing the items user liked in the past. In the Fab system [10] the user's profile is developed and maintained by a personal selection agent.

Document modeling is also done in a similar way by extracting bags of words for each document. One way is by using categorization techniques which classify documents into predefined categories. The classifier determines whether a document belongs to a particular category or not [95] [11]. Some other approaches try to predict the degree of relevance of a document to a user's profile [91] [38]. The degree of relevance is computed through similarities between the vectors of words for both the document and the profile. A similarity score can be computed by different measures - Euclidian distance, Pearson correlation, and cosine similarity are some of the well known methods.

Personalized Recommender System (PRES) creates hyperlinks for a web site which contains pieces of advice about home improvement and makes it easier for a user to find interesting items. System makes these recommendations by comparing a user's profile with the content of each document in the collection. The contents of a document are represented by a set of terms. The user profile is also represented with a set of terms by analyzing the content of documents that the user found interesting in the past. User's interest can be determined through implicit or explicit feedback. Explicit feedback requires a user to provide feedback about documents . On the other hand implicit feedback is recorded by observing the users actions, e.g., clickstream information [91].

PURE [142] a PubMed article recommendation system, is based on content-based filtering. In PURE web-based system users can add/delete their preferred articles. Once articles are registered, PURE then performs model-based clustering of the preferred articles

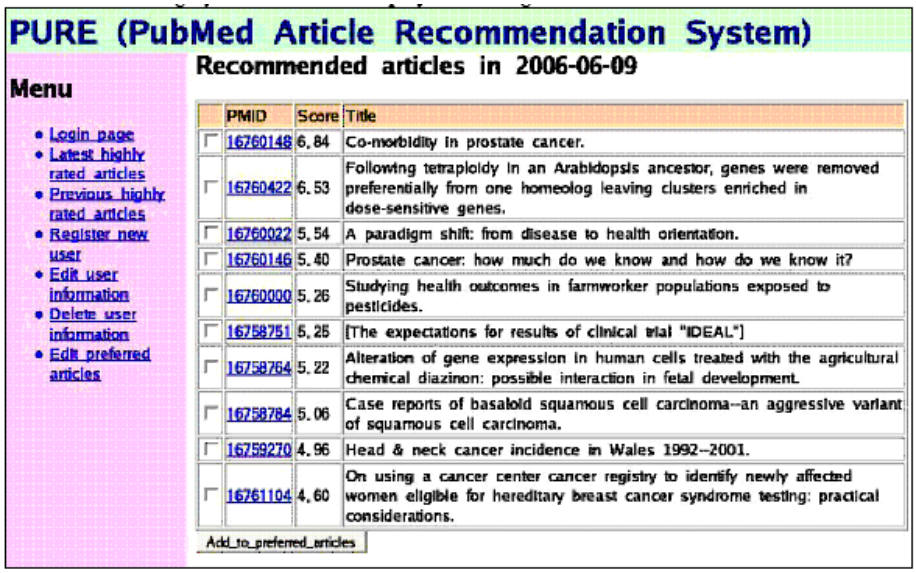

Figure 2.3 Screen shot of PubMed System showing an implementation of content based system.

and recommends the highly-rated articles by the prediction using the trained model. Model-based clustering assumes that the data were generated by a model and tries to recover the original model from the data. This trained model then defines clusters and an assignment of documents to clusters. PURE updates the PubMed articles and reports the recommendation by email on daily-base. This system reduces the time required for gathering information from PubMed. Figure 2.3 shows a sample screen of PURE system which uses content based approach.

Content-based systems provide several benefits over other recommendation techniques. Some other techniques utilize social filtering in which a system maintains preferences of individual users. A recommendation request from a user is fulfilled through finding other users whose preferences co-relate significantly with this user, and system recommends other items preferred by like minded users [41] [124] [108]. These approaches assume that system has sufficient number of preferences from users and preferences of one user match with several other users. In this scenario objects that have not been rated by enough users cannot be recommended. Hence, these approaches generally tend to recommend objects that are popular among other users. Furthermore it is impossible for these approaches to

recommend items that no one has yet rated or purchased. Similarly, when a user first start using these systems, they have no ratings on record hence, if it impossible to find any correlation with other users.. In general, these issues collectively are cold-start problem in which a recommender system cannot make effective recommendations due to an initial lack of ratings [119]. Content-based systems on the other hand uniquely characterize each user without matching his preferences to anothers preferences. Therefore these systems require the analysis of items that one independent user has seen and does not require other users input. Content-based approaches do not have a cold start problem as faced by other approaches. It means reliable recommendation can be created only when the system has the exact knowledge about the users' preferences and needs [4] [60] Objects are recommended based on information about the object itself which can help in providing explanations about what caused an item to be recommended, potentially giving users confidence in the system and feedback about their own preferences [95].

The content-based approach to recommendation has its roots in the information retrieval (IR) community, and consequently inherits many of their limitations. Due to the diversity of resources on the web, not all of the objects could be properly represented using traditional IR techniques. The retrieval of the information from the text document is comparatively easy than other types of objects (images, audio/video etc.) [10]. Also the textual representations capture only one aspect of the content (text), but ignore many others that would influence a user's experience. For example, IR techniques completely ignore aesthetic qualities like readability and all multimedia information. Content-based systems also face specialization problem, it means that the items suggested to the user will be very similar and the customer can be bored by the continuous watching of the documents with overlapping content. Another problem is objects that do not have the exact features specified in the user's profile may not get recommended even if they are similar to user's interest [119] . Collaborative filtering generally leads to more different items that are equally valuable which is referred to as novelty or serendipity of these CF systems [46].

In CB system cross-genre recommendations are not possible for example, a user can only get recommendations very similar to what he / she is looking at and it limits the system to provide recommendations from any other categories. For example, in CB systems normally users cannot get movie recommendation while buying or viewing a book.

Collaborative Filtering: Collaborative Filtering is a process of filtering or evaluating items using the preferences of other people. Work on collaborative filtering (CF) started in early 1990's, but it takes its roots from something humans have been doing for centuries sharing opinions with others [119]. One of the major advancement in the area of information filtering was the initiation of a filtering system by Goldberg et al. Their system Tapestry became the first known recommender system which was based on collaborative filtering technique [41]. Tapestry mail system developed at the Xerox used user reactions to the documents they read. It then used these reactions to filter incoming streams of electronic documents. In this way these systems result in filtering items for a user that similar users filtered. Collaborative filtering introduced a major shift in information filtering research and since been applied in many publicly available systems, and even some commercially available systems.

The basic idea of CF-based algorithms is to provide item recommendations or predictions based on the opinions of other like-minded users without using any descriptive data about items as compared to content-based systems. Few studies showed that collaborative recommender systems can be more accurate than content-based even without using the descriptive data [20] [6]. CF is domain independent in that it performs no content analysis of the items in the domain. Rather, it relies on user opinions about the items to generate recommendations. The opinions of users can be obtained explicitly from the users or by using some implicit measures.

Collaborative filtering methods have been widely used in academia as well as in industry. The electronic mail was one of the first areas where CF was used [42]. GroupLens,

developed at University of Minnesota, was also amongst the first systems that utilized this technique to calculate the correlation between the users of Usenet newsgroups automatically [107]. Implementation of a networked system called Ringo was also based on this technique, which makes personalized recommendations for music albums and artists [124]. Another study showed that the video recommendations provided by a CF system were highly effective [47]. Currently this technique is being used in many commercial applications. Amazon and Moviefinder use Collaborative filtering technique to recommend products to their customers. In these systems users have to provide ratings for different products and this rating information is used recommend other product that might be of interest to the user. Ratings are provided by the user manually hence, these systems are not considered fully automatic. CDNOW is a fully automatic system, user behavior is inferred by the actions of a customer on the CDNOW website. General recommendation engine (GRE), developed for National Science Digital Library collections, is also a fully automatic system in which users clicks are stored and processed to recommend documents using association rules [136].

The output of a collaborative filtering algorithm is a list of items for a particular user based on the user's previous likings and the opinions of other like-minded users. Like-minded users are selected using the history / rating data and finding the correlation amongst users. In a typical CF system, there is a list of m users and a list of n items. Each user has a list of items which the user has expressed his opinions about. Opinions can be explicitly given by the user as a rating or can be implicitly derived from purchase records, by analyzing timing logs, by collecting web hyperlinks etc. There are number of collaborative filtering algorithms that can be divided into two main categories user-based (memory-based) and item-based (model-based) algorithms [17].

User-Based Correlation: User-based algorithms utilize the entire user-item database to generate a prediction. It is an information filtering technique that use group opinions to recommend information items to individuals [107] [63]. User-based correlation utilizes the

correlation between a particular user and other users who have purchased or liked an item. These systems use the history / rating information to find out neighbors of a given user. These neighbors have either rated various items similarly or they tend to buy similar sets of items as the given user. Once a neighborhood of this user is formed, rating / history information of these users is used to produce a prediction or top-N recommendation of unseen objects for the active user. This top-N recommendation list is ranked in the order of relevancy. Relevancy can be computed using various methods and the most famous are cosine similarity and pearson correlation. This technique, also known as nearest-neighbor or people to people collaborative filtering is very popular and widely used in practice.

User-based correlation systems are persistent since learning about patterns between users requires substantial data which is collected over time. However, these systems can be automatic as well as manual. An automatic system does not require users to provide any information explicitly. Browsing / click-stream history information is stored implicitly and latter used it to produce recommendations. On the other hand in manual system users have to explicitly rate the products and this rating information is used to compute the recommendations [120].

User-based collaborative filtering systems use the opinions of a community to recommend items to individuals. In the music example, a collaborative filtering recommender would identify other people who share your music tastes, and would then recommend to you music that those "neighbors" liked but that you hadn't yet heard. MovieLens utilizes user-based collaborative filtering technique and matches preferences of a user with the preferences of other users with similar movie preferences [63]. My CDNOW is a system that uses user-to-user correlations to identify a community of customers who tend to own and like the same sets of CDs. New music recommender iLike suggests songs you might enjoy based on your listening habits and other users with similar tastes. The principle is that if several members of a community owned and liked the latest album, then

Figure 2.4 iLike uses user-based collaborative filtering approach.

it is highly likely that another user from this community will also like it. Figure 2.4 shows music recommendations from iLike system.

User-based algorithms face sparsity and scalability problems. Sparsity problem arises because normally there are large numbers of items in commercial recommender systems like Amazon.com and even active users may have purchased less than 1% of the items. This results in poor accuracy of the recommendations. Scalability is another issue which will arise when the number of users and items will increase. With millions of users and items it will be time expensive operation to compute the recommendations. Item-based correlation approach can solve the sparsity and scalability issues.

Item-Based Correlation: Unlike the user-based collaborative filtering algorithm the item-based approach looks into the set of items the target user has rated. System identifies items frequently found in "association" with items in which a customer has expressed interest. Association may be based on co-purchases, ratings by common customers, or other measures. Using this information item similarity is computed, for example, similarity

between object i and j is computed by first isolating all the users who have rated / purchased both items and then applying a similarity measure like cosine similarity or Pearson correlation. Once the most similar items are found, the prediction is then computed by taking a weighted average or regression value of the target user's ratings. Final list is ranked in the order of regression values computed.

Item-based correlation takes a probabilistic approach and the collaborative filtering process computes the expected value of a user prediction, given his ratings on other items. Item-based collaborative filtering scales to massive datasets and produces high quality recommendations. Rather than matching the user to similar customers, item-to-item collaborative filtering matches each of the user's purchased and rated items to similar items, then combines those similar items into a recommendation list [76]. Amazon.com's recommender system is one of the famous item-based CF systems. Amazon.com uses a two stage process to generate recommendations. In the first offline stage the algorithm builds a similar-items table by finding items that customers tend to buy together. As this process is done offline, systems scalability issue does not arise. In the second stage this algorithm finds items similar to each of the user's purchases /ratings, aggregates those items, and then recommends the most popular or correlated items. This process is very quick as it only depends on the number of items the user purchased or rated in the past. Figure 2.5 shows a sample list of recommendations from Amazon.com using item-based correlation approach.

Collaborative filtering solves several limitations in content-based filtering techniques [10]. Due to the diversity of resources on the web, not all of the objects could be properly represented which limit the implementation of content-based system in these domains. For instance, retrieval of the information from the text document is comparatively easy than other types of objects (images, audio/video etc.) [10]. Collaborative filtering on the other hand do not make use of object features rather uses opinion of other users to make recommendations and can be effectively used with any type of object collection. Similarly, textual representations capture only one aspect of the content (text), but ignore many others

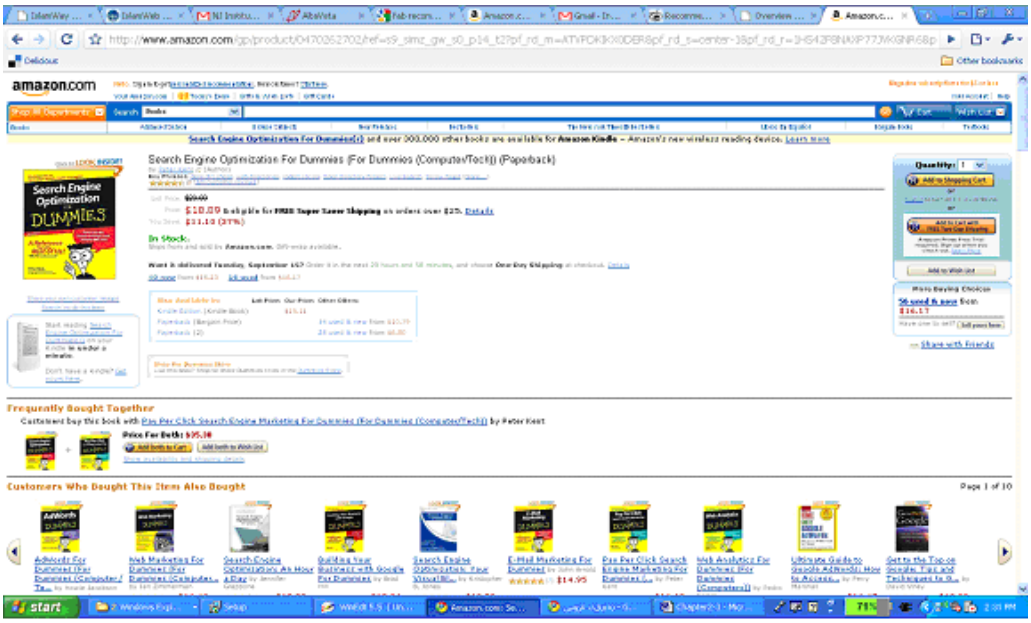

Figure 2.5 Amazon uses item-based collaborative filtering method.

that would influence a user's experience. For example, IR techniques completely ignore aesthetic qualities like readability and all multimedia information. Collaborative system can effectively make use of aesthetic properties noted by other users with similar interests thus can effectively overcome this issue. Content-based systems also face specialization problem, it means that the items suggested to the user will be very similar and the customer can be bored by the continuous watching of the documents with overlapping content. Collaborative filtering generally leads to more different items that are equally valuable which is referred to as novelty or serendipity of these CF systems [46]. In CB system cross-genre recommendations are not possible for example, a user can only get recommendations very similar to what he / she is looking at and it limits the system to recommendations from any other categories. On the other hand collaborative filtering can provide cross-genre recommendations based on the opinions of likely minded users.

Despite being a successful technique in many domains, CF has its share of shortcomings. One of the major issues with collaborative techniques is cold-start problem. Cold-start problem means system has no way to recommend a new item to users or to provide an accurate predictions for a new user. Due to cold-start problem collaborative

recommender system cannot make effective recommendations [119]. Collaborative approaches process recommendations by finding users whose preferences correlate significantly and recommends items preferred by likely minded users [41] [124][108]. These approaches assume that system has sufficient number of preferences from users and preferences of one user match with several other users. In this scenario objects that have not been rated by enough users cannot be recommended. Hence, these approaches generally tend to recommend objects that are popular among other users and cannot effectively utilize the whole item base. Furthermore it is impossible for these approaches to recommend items that no one has yet rated or purchased. Similarly, when a user first starts using these systems, they have no ratings on record hence, if it impossible to find any correlation with other users. Content-based filtering is based on the document features and as such does not face such cold-start problem. Condliff et al. proposed a Bayesian methodology for recommendation system which uses Bayesian theory to give a good prediction by fully incorporating all of the available data to cop with the cold-start problem [31]. Claypool et al. also proposed an approach to solve cold-start problem what was based on a weighted average of the content-based filtering prediction and collaborative filtering prediction [30].

Another challenge that collaborative filtering systems face is sparsity problem which is due limited ratings or opinions from users. It is very difficult to convince users to provide their opinions explicitly. Since these systems depend on the votes of users compute the similarities among users, it is very important to get enough opinions from the users. Due to this reason usually in collaborative systems, the number of ratings already obtained is usually very small as compared to the number of ratings that need to be predicted. For example, in the movie recommendation system, there may be many movies that have been rated by only few people and these movies would be recommended very rarely if they were rated highly. Also, for the user whose preferences are uncommon, it is highly likely that there may not be any other similar users, which result in poor recommendations

[10]. Maintaining and using a profile can help mitigate the sparsity problem. In this way two users could be considered similar if they have similar profiles even if enough rating information is not available. Pazzani used the gender, age, area code, education, and employment information in the profile to compute similarities amongst users [101]. This type of filtering techniques is also referred to as demographic recommender system [101]. In another study sparsity issue was handled by implementing associative retrieval framework and activation algorithms which helps to determine transitive associations between users by their transactions and feedback [52]. In some other studies Singular Value Decomposition (SVD) was used to cop with the sparsity problem by reducing the dimensionality of sparse ratings matrices [117] [92]. Although explicit feedback mechanism e.g., rating information, leverage the calculation of similarity, implicit feedback is usually easier to record and more helpful to decrease the sparse matrices. Implicit methods can be implemented by monitoring user's behavior or user's browsing time on the page. Clickstream information or browsing time shows users interest in a particular website or object. Explicit methods require users to provide input in the form of ratings, voting or opinion in order to provide recommendations. The system also can use compensation methods in which user only gets recommendations after providing some rating information.

Another challenge of conventional collaborative filtering algorithms is the scalability issue [118]. As the amount of information increases quality of recommendation becomes better but it affects the efficiency. With million of users providing rating on hundreds of thousands of items create huge matrices which results in scalability issues for collaborative filtering systems. These approaches often cannot cope well with the large numbers of users and items. The model-based collaborative approaches alleviate scalability issues through upfront computation but these approaches tends to limit the range of users [141].

Knowledge Based: One of the major shortcomings of the CF or CB techniques is that they cannot provide recommendations with a holistic view of the domain as they cannot

provide explanations about why the recommended objects are relevant to the user within the domain. Knowledge based approaches work by mapping users' needs to product features in order to provide more personalized recommendations. These systems gather information on the requirements of desired products, and tries to map the user information to appropriate item descriptions [29]. These types of recommender systems are able to reason about the relationship between the user need and a possible recommendation [20].

Although prior research has shown content-based and collaborative filtering as successful techniques to aid users but there is emergent requirement of more personalization. Knowledge-based recommender systems can generate more personalized recommendations while taking into consideration background knowledge of the users. These systems map user needs to the products and suggest them to the users. Knowledge-based approaches utilize the functional knowledge: about how an item can meet a specific user need, and can therefore reason about the relationship between a need and a possible recommendation. Knowledge-based recommender systems do not generalize user base, rather match user's need with the set of available options. These systems do not incur ramp-up and sparsity problems because they do not use statistical evidence to provide recommendations [20]. Schafer et al. call Knowledge-based recommendation the "Editor's choice" method [120].

Knowledge-based recommender systems provide recommendations based on inferences about a user's needs and preferences. Although all recommendation techniques use some kind of inference, however, Knowledge-based systems have also functional knowledge not used in other techniques. This functional knowledge tells how a particular item meets a specific user need. This functional knowledge is either entered manually by experts like in case of Entre system or acquired automatically by computer programs. For example, in recommendation engine's KB system, this functional knowledge is extracted by running naive bayes classifier. Functional knowledge helps in reasoning about the relationship between a need and a possible recommendation. This functional knowledge is

stored in a user profile and it can be any knowledge structure that supports this inference. The knowledge structure could be very simple, as in Google, uses information about the links between web pages to infer popularity and authoritative value [19]. In other cases it can be more representative of the user's needs. In such cases it requires the additional information about users and products that gives the system power in difficult recommendation tasks, and also controls both the ramp-up problem and the sparsity problem experienced by current recommender systems [133]. The Entree system and several other recent systems employ techniques from case-based reasoning for Knowledge-based recommendation. Entree uses knowledge of cuisines to infer similarity between restaurants (Robin Burke). Entree is an interactive system that recommends restaurants to the user based on factors such as cuisine, price, style, and atmosphere, etc. or based on similarity to a restaurant in another city. The user can then provide feedback such as finding a nicer or less expensive restaurant and the system will refine the results according to user input.

Output of a Knowledge-based system is a ranked list of items as in other recommender systems. However, this ranking process is different than other systems. In Knowledge-based system a score is given to each particular item's feature that meets any of the user's need. Scores for each item are combined which shows how well a particular item fulfills user's need. In the end items are ranked from the highest score to the lowest and top-k are presented to the user.

Knowledge-based system does not involve a start-up period during which its suggestions are of low quality, therefore these systems don't have ramp-up or cold start problem as in collaborative filtering systems. A Knowledge-based recommender cannot find user groups, the way collaborative systems can, however, it can make recommendations as wide-ranging as its knowledge base allows.

Although Knowledge-based systems provide certain benefits over content-based and collaborative filtering systems, however, they also have limitations of their own. Primarily these systems have disadvantage due to limitations in acquiring knowledge. These systems

require three types of knowledge, Catalog knowledge, functional knowledge and user knowledge, to provide recommendations. Catalog knowledge is about the items being recommended and their features. Functional knowledge helps in mapping user's needs with the object that might satisfy those needs. User knowledge is required to provide good recommendations. This knowledge could be in the form of demographics or specific information about the need. User knowledge is the most challenging and in the worst case it is an instance of the general user-modeling problem [133].

Hybrid: Hybrid recommendation approaches combine techniques of other type of recommender systems. The main goal of hybrid approach is to avoid the shortcomings of the other enumerated methods and get advantage from their benefits [4] [94] [101]. Usually techniques from content-based and collaborative filtering are combined to develop hybrid systems. There are several possible ways to combine different recommender techniques in order to develop a hybrid system [4]. It can be done by implementing various methods separately and combining the outputs of these methods [136]. The outputs received from individual systems can be combined using a linear combination of ratings [30] or a voting scheme [101]. Another way is by combining characteristics of different approaches into one system and in this way approaches complement each other and contribute to the others effectiveness [94]. Melville et al. proposed a hybrid approach in which a collaborative filtering based on users' ratings is supplemented with more ratings obtained via content-based predictor [89]. Fab is based on collaborative filtering but also maintain the content-based profiles for each user [10].

Many hybrid recommender systems have been successfully built in the past. TechLens, a hybrid recommender algorithm, successfully combined Collaborative Filtering and Content-based Filtering to recommend research papers to users. This algorithm combined the strengths of each filtering approach to address the individual weaknesses [132]. Stanford University digital Library system Fab recommended Web pages by choosing neighbors for

CF-based recommendations using CB-based user profiles. Combining both collaborative and content-based filtering systems, Fab eliminated many of the weaknesses found in each approach. Fab's hybrid structure allows for automatic recognition of emergent issues relevant to various groups of users. [10]. Woodruff successfully developed six hybrid recommender algorithms that combined textual and citation information in order to recommend the next paper a user should read from within a single digital book [138]. Hybrid recommender systems combined with Knowledge-based approach can improve recommendation accuracy and overcome some of other limitations of traditional recommender systems. EntreeC system combines knowledge-based recommendation and collaborative filtering to recommend restaurants [20].

Output of a hybrid system is a ranked list of items. This ranked list can be formed by using the individual lists from each type of recommender system and then aggregating the results. This list integration operation can be done by simple aggregation function like max, min, avg. or more complex functions like skyline etc. In case of general recommendation engine [136] this list was formed using simple aggregation function.

Currently hybrid recommendation systems have been used by many commercial website like netflix.com. The Netflix Web site makes recommendations automatically using a system called CineMatch. CineMatch uses information from various sources to determine which movies customers are likely to enjoy. These sources include films themselves, which are arranged as groups of common movies similar to content based systems. Customers' ratings, rented movies information, current queue and the combined ratings of all Netflix users are also used in the process of making recommendations from cinematch. Collaborative systems commonly use such sources of information to make recommendation. Figure 2.6 shows a sample screen for netflix recommendations.

Several researchers have compared the performance content-based and collaborative approaches alone with hybrid approach and showed that the hybrid approaches can be more accurate then both of these individually. [10][89][101][129].

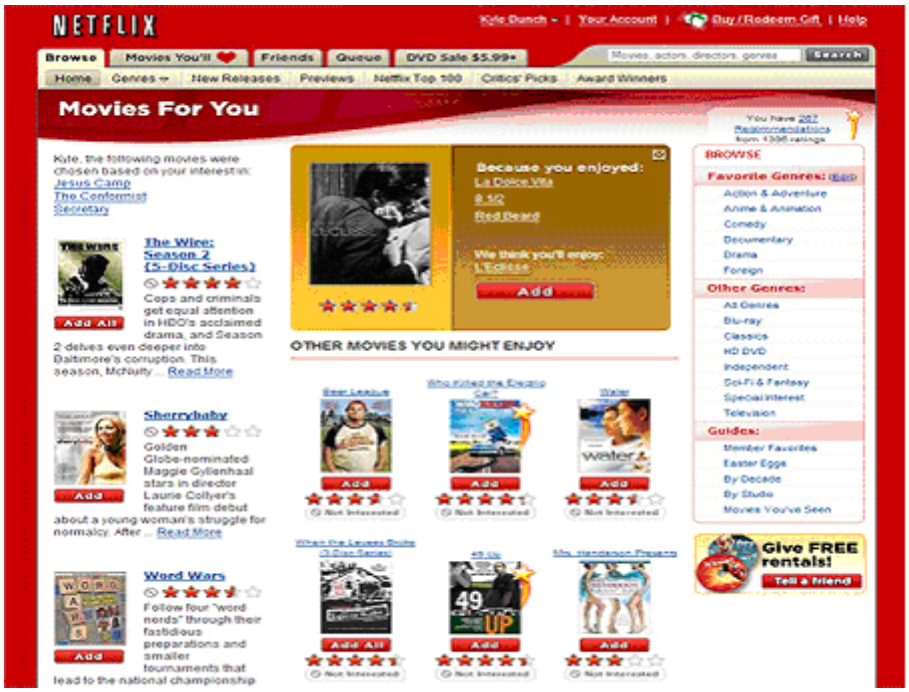

Figure 2.6 Netflix uses hybrid approach to process recommendations.

2.1.3 Performance Challenges

Recommender systems face performance challenges in terms of effectiveness of the recommendations as well as how efficiently these recommendations are being computed.

Effectiveness: Recommendation effectiveness has been a major concern for researchers since the beginning of research in this field. Several recommendation techniques have been proposed to cop with this problem. Over specialization , and limited feature extraction capabilities result in effectiveness issues for content-based systems where as sparsity, and cold-start problems are major hurdles to produce quality recommendations using collaborative systems. Effectiveness of a recommendation technique is usually measured through coverage and accuracy metrics. Coverage measures the percentage of items for which a recommender system can make predictions [46]. Coverage can be measured generally through statistical or decision-support [46] methods. Statistical methods generally compare the estimated ratings with the actual ratings in the user item matrix. Decision-

support methods measure how well a recommender system can make predictions about highly relevant items. Most of the research studies in recommender system domain have focused on improving the quality or effectiveness of recommendations. Inception of various recommendation techniques is primarily due to the effort of producing effective recommendations that can help users.

Efficiency: In recommender systems the result of a query is a ranked list of items based on relevancy score. Generating ranked lists is typically an expensive operation that often results in access latency. This is especially problematic when the volume of data is extremely big. This problem is often referred as scalability issue [118]. As the amount of information increases quality of recommendation becomes better but it affects the efficiency. With million of users providing rating on hundreds of thousands of items create huge matrices which results in scalability issues. These approaches often cannot cope well with the large numbers of users and items. Another problem is due to the presence of these matrices on the secondary storage devices. These devices are slow in access and result in access latency. Model-based CF approaches are developed to cop with the scalability problem to some extent. How these approaches can only provide limited performance improvement in terms of efficiency. With the increasing popularity of the World Wide Web, amount of information and number of users are growing exponentially. This increase has created challenges for developers to provide efficient solutions. Traditional applications such as file transfer, news and email need more throughputs but can tolerate delays. However, applications of interactive nature like recommender systems require latencies on the order of seconds [15]. Recommender Systems, being computationally intensive and interactive applications, cannot tolerate access latency. Although advanced as well as more powerful physical resources can help improve the performance yet there is dire need for further improvement by using other optimization techniques.

Caching is one of the optimization techniques and it has seen significant success in reducing latency in storage systems [62][65][72] and in processor memory hierarchies [128]. These techniques can potentially be used to cop with the efficiency problem in recommender systems. Web-Caching is one of the optimization techniques to improve the efficiency of web applications. Cache is a temporary storage area where popular or frequently accessed data is stored for rapid access. Once the data is stored in the cache, future use can be made by accessing the data in the cache, rather than re-fetching or recomputing the original data. This results in reducing the average access time and improving the response time. Caching of results was noted as an important optimization technique in Google search engines [19]. There has not been any explicit study on caching for recommender system applications however, this approach is very much applicable for any type of recommendation system.

2.1.4 Summary

Information overload is becoming more and more complex with the rapid growth of web and recommender system is a viable solution to cop with this problem. Recommender systems assist users in finding desired information and making decisions about product and services. Recommendations can be generations using several techniques where as most popular are content based and collaborative filtering approaches. The former generates recommendations based on the similarities of content while the later provides recommendations based on users' evaluations and preferences [75]. Both of these approaches are being widely used but have their own limitations in providing quality recommendations. Knowledge-based systems avoid some of the limitations in content-based and collaborative systems but have its own disadvantages. Hybrid techniques compute recommendations by combining multiple approaches into a single system [10]in order overcome the disadvantages of individual systems. Figure 2.7 summarizes the advantages and disadvantages of each of these approaches.

Recommendation Approach	Advantages	Disadvantages
Content-Based	• Rating / history information not needed • No ramp-up required • No sparsity problem	• Require descriptive data for each object • Specialization problem • Cross-genre recommendations not possible
Collaborative Filtering	• Provide cross-genre recommendations • Descriptive data not required	• New user ramp-up problem • New item ramp-up problem • History / rating information needed • Sparsity problem
Knowledge-Based	• Map from user needs to products • No ramp-up required • Rating / history information not needed	• User profile / preferences information required • Knowledge engineering required
Hybrid	• Combine advantages of multiple approaches	• Resultant list might ignore useful recommendations from individual systems

Figure 2.7 Table summarizing the benefits and limitations of famous recommendation approaches.

Generating recommendation lists requires an enormous amount of resources that often result in access latency problem. However, applications of an interactive nature like recommender systems require latencies on the order of at most several seconds [86].Sites risk losing patronage due to the frustration of users by slow access to information. Caching frequently accessed data has been a useful technique for reducing stress on limited resources and improving response time. Web caching is one of the optimization techniques which is successfully used to improve the efficiency in various applications and in the next chapter provides literature review of caching.

2.2 Caching

Recommendation techniques are found to be very effective in mitigating the information overload problem. No matter, how effective any such application is, poor efficiency can result in non-usage of the system and ultimately leads to the failure of such system. Poor efficiency is always a big threat in the success and usefulness of any system. Many techniques have been emerged in the recent years to cop with the problem of efficiency in web based systems.

Research for improving performance of web based systems can be divided into two areas: improvement in web servers and improvement in networks. Previous work in many studies has focused on techniques for improving server performance. These studies often suggest improvement to application and the operating systems. Research in the area of network has focused on improving network infrastructure performance for Internet applications. Researchers have used web caching in both areas to improve the performance of web based systems. It is a technology targeted to reduce the transmission of redundant network traffic, server load and access latency.

Cache stores popular or frequently accessed data for rapid access. Once the data is stored in the cache, future use can be made by accessing the data in the cache, rather than re-fetching or recomputing the original data. This results in reducing the average access time and improving the response time. Caching of results was noted as an important optimization technique in Google search engines [3].

Caching has been effectively used to decrease the delay of access to data available over the web. It helps reduce the page generation delay in generating the web pages requested by the web users. Also local Cache helps in reducing the network latency where request to a document on the network results in a cache hit on the local machine.

2.2.1 Caching Overview

A cache retains a copy of data which is computed earlier or stored some where else and is costly to fetch or to compute when compared to the cost of fetching from the cache. In general, a cache acts as a temporary storage place where frequently accessed data is retained for quick access. After the data is available in the cache, future requests for this data are fulfilled by accessing this cached data rather than re-fetching or recomputing the original data and it helps to decrease average access time.

Caching concept was first introduced in 1967 as an exciting memory improvement in Model 85, a latecomer in the IBM System/360 product line [104]. Since then it has

been used to improve efficiency in various domains. CPU cache helps expedite data access that the CPU would otherwise need to fetch from main memory. The page cache in main memory is usually managed by the operating system kernel helps improve the performance of main memory. Database caching can substantially improve the throughput of database applications. Web caching deals with problems such as redundant data transmission, limited bandwidth availability, slow response times, and high costs and target to improve performance in these areas.

Caching techniques have seen significant success reducing latency in storage systems [62][65][72] and in processor memory hierarchies [128], it remains to be seen how effective such techniques can be within the World Wide Web.

2.2.2 Cache Location

Proxy Caching: Proxy caching, also known as forward proxy caching, are usually deployed by internet service providers, schools and corporations to save bandwidth. A proxy server gets HTTP requests from users, matches it with the information inside the cache, if found sends the requested object back to the user. If not found in the cache, a request is sent to the origin server on behalf of the user. The object is then fetched from the origin server, a copy of it is may be retained in the cache before sending it to the user. A careful placement of proxy caches can lead to bandwidth savings, quicker response times, and enhanced accessibility to static web objects [98].

A proxy server can be placed in the user's local computer or at specific key points between the user and the destination servers but normally it is positioned at the edge of a network in order to serve large number of internal users. In this type of approach users have to manually setup the appropriate proxy for use. One of the drawbacks of this type of caching is that it is a single point of failure in the network. Using proxies between clients and servers reduces bandwidth usage, server load and reduces user access latency

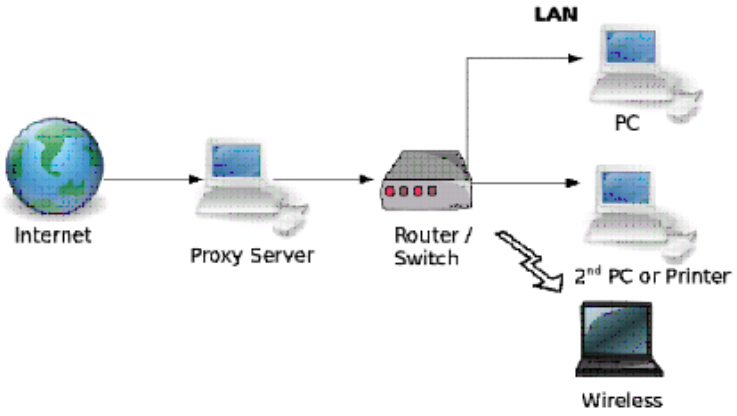

Figure 2.8 Figure showing the architecture of proxy caching.

[37][85][111]. However, proxy caching has significant overhead in placing dedicated proxies among the Internet. Figure 2.8 depicts a proxy server.

Browser Caching: Browser Caching is implemented on the user side. Most Web browsers facilitate caching information in the memory or hard disk of user's machine which helps in reducing the response time [139]. It keeps a copy of visited pages on the local disk for later retrieval. It checks to make sure that the objects are fresh, usually once a session. This cache is useful when a client hits the 'back' button to go to a page they've already seen. Also, if same navigation images are used throughout any website, browser cache keeps a copy of it and sends it immediately on subsequent pages. User side caching reduces access latency significantly because it helps in answering the repeated request from the cache. It not only reduces the access latency but also server load and network traffic. Web servers have limited resources and the benefit of the sum of individual user's gain is huge. However, user side caching cannot take the benefit of shared caching. If most queries are shared and not repeated by the same users then these queries cannot be answered from the cache.

Chen et al. have proposed a browser-level web caching system [24]. The system supports hybrid and cooperative caching and it is based on chord. The nodes on the network contacts with other nodes for the sharing of URL based web caching. The proposed model

takes advantage of URL and static web object .As their system is based on chord it is benefitted from advantages of chord. Chord provides the simplicity and proven performance boost to the proposed system. Through the performance analysis of their model they have been able to demonstrate that browser-level web caching system can improve the hit rate.

Server Side Caching: Server side cache also known as reverse proxy cache is helpful for servers that anticipates a substantial number of requests and need to maintain a superior quality of service in terms of response time. This type of caching is common in database backed web applications. When a query is sent by a user, it is first checked in the cache and if available, answer is sent back to the user. In case of cache miss answer is requested from the server as shown in Figure 2.9. Server caching provides several benefits. It has small overhead as compares to proxy caching because one resource can answer the entire incoming requests. Moreover, it allows the maximum query sharing among different users and the hit rate would be high by caching popular queries.

Although server caching provides benefits but there are certain issues and challenges that must be addressed. An important issue is how to maintain consistency between the original web page at the server and the cached web page. With the increasing number of dynamic web sites, the probability of reusing the cached content decreases. Applications, like recommender systems, are connected to databases for producing recommendations dynamically using data retained in databases. If the data is highly dynamic e.g., stock data, news items, the validity of a cached page could be very short and the probability of successfully reusing the cached pages will be low. Recommender Systems attempt to predict items (movies, music, books, news, Web pages) that a user may be interested in. In such case actual data at the server could change as new items can arrive. Also, as in case of recommender systems, answer for various users could be diverse; hence, probability of using the cached pages would be low.

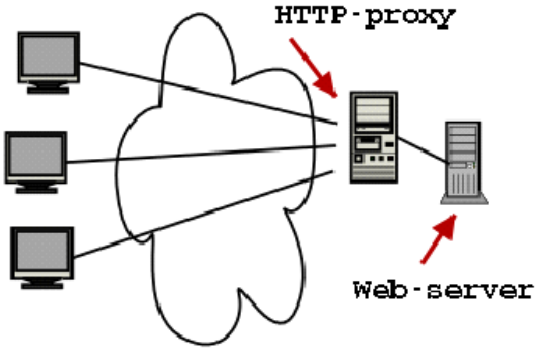

Figure 2.9 Figure showing an architecture of server side cache.

Zeng and Veeravalli have proposed a novel server side web caching model called H/T and Hk/T for multimedia application [143]. Their proposed model provides flexibility to choose different objects. Later on these objects could be place and replaced in proxies. Apart from hit rate the model also considers the multimedia object sizes and their playback time as an important factor. This model is the first one that provides cooperative server-side caching strategy, for multimedia systems. The experimental tests to evaluate the performance of this model have demonstrated that the proposed model has outperformed one of the most popular Least Frequently Used (LFU) algorithm significantly and it can improve the performance of very large multimedia.

Kumar et al. proposed a model called Object Caching Service (OCS) to save the results of invocations of reading of objects on the server side [68]. The model is based on CORBA and provides strong consistency using dependency graph techniques. Their model is implemented as a CORBA service. The model aims at caching of the dynamic contents and its Caching Service is implemented by using CORBA. Client requests are intercepted by the help of CORBA intercept services. Objects register themselves to OCS with the help of a specifically designed registration interface. To process an intercepted request OCS checks the validity of request in cache. If the request is found in the cache, the result is served from the cache to the clients otherwise the results are saved in the cache first and

then are sent to the client. They have used a simple policy for cache replacement to replace the least used objects when cache is full.

Menasce et al. proposed a model to improve the performance of online auction system by using server side activity based caching [90]. The model tackles the problem of server overloading which may delay the bids and some of bids reach to the server after the biding is closed. This may reduce the revenue of the auction sites as some of the highest bids may not reach to the server. They have proposed a server-side caching model that employs policies that are based on auction-related parameters these parameters include number of bids placed before the end of biding time. Through their analysis they were able to show that auction sites can increase the performance with very small caches rather than using the larger one. These small caches produce high hit rates when compared with large caches. The proposed model is implemented using a three tiered auction site and the results collected from this implementation indicated that the proposed model has increased the performance of auction system significantly. At the end they were able to prove that server side caching can improve an auction system considerably.

Saleh et al. proposed a model to serve dynamic web object to the client efficiently [113]. Their model enables users to do internet surfing faster and reduces the load on the server. The model is based on the "URL Rewriting" feature. This feature is provided by some of the most popular web servers these web servers includes Apache that provides a built-in feature of caching of dynamic objects. The model provides control of cached data to the system administrator that was not the case of proxy and browser caching. They have evaluated the performance of their model by using a simulated environment. The results obtained have shown that the model has reduced the response time of dynamic web pages considerably.

Zhang et al. has developed a novel server side caching system [144]. Their model provides an efficient way of storing and retrieving of cached objects. The model organizes an image map of similar objects in a file by relating them with in-memory data and data

stored on disc. In this way it was able to provide a very efficient way of finding the large cached objects. They have done the performance evaluation using two categories of tests, the first one is to evaluate the hit rate that finds an object in main memory. The second on calculate the number of time it has to access the hard disc to find and load an object if it is not found in the memory. The evaluation is done on a simulation system created to address the above mentioned test categories. The Results have shown that the proposed model has very high hit rate and has almost no need to access the objects on the disc. The response time to access objects on disc was also very good.

Shen has presented new concepts of meta-caching and metatranscoding [125]. In this model the intermediate results are cached and the future identical requests served through transcoding from the metadata that was cached earlier. The model reduces a lot of computing load on the server. The results indicate that by using the proposed meta-caching method, even a small size cache could decrease a significant amount of computation cycles. It has also improved the start-up latency for memory based implementation.

2.2.3 Web Traffic Characteristics

Several research studies have shown that people browsing behaviors show certain kind of pattern. Studying this pattern allows us to manage the cache data effectively. There are two most common laws that state the web traffic characteristics e.g., zipf's law and locality of reference.

Zipf's Law: Zipf's law states that the relative probability of a request for the ith most popular page is proportional to 1/i. People tend to access the same piece of information over time, more specifically the usage pattern follows the Zipf's law. Breslau et al. used six traces from proxies at academic institutions, corporations and ISPs and found that the distribution of page requests generally follows a Zipfs like distribution [18].

Caching solutions require the selection of objects with higher probability of being accessed in the future. One rule for choosing such objects is through Zipf's law, which has been applied in a number of areas. It predicts that the probability of access for an object is a function of its popularity. [123] Verified that high cache hit rates can be achieved using Zipf's law caching. Markatos found out that some queries are very popular and a large number of queries were requested repeatedly which can be effectively used for caching purposes [86]. Query repetition frequency follows a Zipf distribution [139]. Several researchers have observed that the relative frequency with which web pages are requested follows Zipfs law [40] [32]

Locality of Reference: Locality of reference suggests that an application does not access all of its data at once with equal probability but rather exhibits temporal and/or spatial locality. Former suggests that if some data is requested, then there is a high probability that it will be requested again in the near future. Later advice that if some data is requested, then there is a high probability that data nearby will also be requested in the near future [14].

Computer memory cache, a specially designed faster memory area, keep both recently referenced data and data near recently referenced data for caching purpose. Similarly, in web applications effective use of temporal and spatial reference can help implementing the web cache. When the number of users increase, the locality of reference within that group gets stronger, and caches become more effective. Several studies have indicated that a significant amount of locality of reference exists in the queries of web search engines. Log traces of Alta Vista search engine showed almost 33% queries were repeated by same or different users [127]. In another study Markatos examined the query traces of EXCITE search engine and found out that a large number of queries were repeated [86].

Although these traces show a significant number of repeated queries however, it only quantifies the locality of these queries if the same query is repeated in a short interval.

In order to find out the temporal locality in these submitted queries, the time between submissions of same query at different times was measured in several studies. In Excite search engine traces, for 1,639 queries this time interval was less than 100. So a cache size large enough to keep recent 100 queries can help to answer 1,639 requests. Similarly, traces show that for 14,075 queries the time between repeated query submission was less than 1,000 and for 68,618 queries this time was 10,000. In Excite query traces 83% of the queries were repeated within an hour [86]. In another study Xie et al. observed the Vivisimo traces and found out that 65% of the queries were asked within an hour again. Rizzo et al. observed this property in the web proxy traces collected at University of Pisa, Italy [110], and Cao et al. observed this property in the digital equipment corporation's proxy traces [21].

All these studies show that queries submitted exhibit excellent locality in the traces. Utilizing this property, a significant number of queries can be cached and high hit rate can be achieved. Figure 4 shows a sample characteristic of queries submitted to EXCITE search engine [86].

2.2.4 Cache Selection Policies

Cache servers have a limited capacity for storage of Web contents. Once the cache is full there must be a procedure to replace some of the cached contents with newer ones. The replacement policy for a cache determines which documents to remove to make room for new data to be brought into the cache. Empirical studies have indicated that the choice of page replacement policy for Web caches can have a serious effect on the utility of the cache. Figure 2 shows the cache hit and cache miss scenario. When a query is sent by a user, it is first checked in the cache and if available, answer is sent back to the user otherwise is requested from the sever as shown in Figure 2.10. Although Zipf's law and Principle of Locality show the characteristics of web usage patterns however, these rules cannot help in deciding the exact objects to keep in the cache. Cache replacement algorithms help

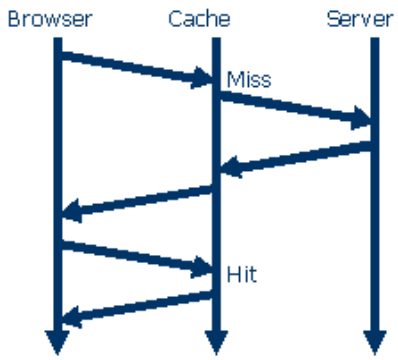

Figure 2.10 Cache hit miss scenario.

in selecting the proper objects. They often aim to minimize various parameters such as the hit rate; the byte hit rate, the cost of access and the latency. The most widely used cache replacement algorithms include static caching, Least Recently Used (LRU), Least Frequently Used (LFU) algorithms and hybrid algorithms. Selection of best algorithm depends on the size of cache, user population, and type of application. As the cache size and user population increases, cache hit rate also increases.

Many classifications of replacement algorithm has been presented. In general, the important factors of Web objects that can influence the replacement process are frequency, recency, size, cost of fetching, modification time, and expiration time [64]. In another study these factors were combined into three broad strategies for cache replacement: recency-based strategies, frequency-based strategies and hybrid strategies [55].

Static Cache selection Policy: Web access patterns change very slowly which leads to static cache selection policy. In static caching a cache is filled with set of objects which help to maximize the cache performance. In static caching content in the cache remains constant for certain period of time thus doesn't result in CPU overhead like other policies [130]. In static cache selection policy a fixed set of object are kept in the cache for a relatively long period of time. This set is determined periodically to maximize the cache hit rate. Once the cache is filled, no objects are replaced in the cache throughout the predefined timeperiod and it is expected that the cached objects will be accessed more frequently than the other documents.

It is known that people tend to access the same piece of information over time, more specifically the usage patron follow the Zipf's law [140]. Search engines use this fact and fill the cache with the most popular queries by analyzing query logs. This type of approach can serve in a static manner if a cache is loaded in batch mode and not modified until the next batch update.

Markatos used this approach using EXCITE query log and showed that static caching policy is a good choice for small cache sizes [86]. Static caching can also be combined with dynamic approaches like LFU, LRU etc. Fagni et al. proposed a static-dynamic in which cache is divided into two parts where one part is used for static caching and the other is used for dynamic caching [36]. Both of these studies measured the query frequency to select the static cache content and then fill the cache with the most frequent queries. Yates et al. proposed an admission policy to select infrequent queries that will not be submitted in the future. Static caching perform well for highly loaded Web servers with a limited cache size [130]. However, static caching may not be effective for dynamic Web sites which provide up-to-date information and results in unstable access patterns. Static caching can be combined with other dynamic policies to provide solutions for such applications.

LFU: As discussed earlier, web access follows zipfs law. This law helps effectively manage the caches by storing popular objects. LFU replacement policy is typically used when data follows zipfs distribution. Arbitrarily high cache hit rates are possible by storing the most popular objects and employing the LFU replacement policy [123][18].

Least frequently used algorithm removes the object that was retrieved least frequently from the cache. LFU policy works well with zifp's like distribution because it keeps the popular objects in the cache and removes the least popular ones. Study results show that, for large enough cache sizes, LFU is optimal and even for smaller caches is better than widely used policies like LRU [18].

Several variations of LFU algorithm have been studied. LFU-Aging keeps objects that were very popular during one time period in the cache even when they are not requested for a long time period. LFU-DA is another policy designed to overcome the limitations of LFU-Aging due to the heavily dependency on parameters [9]. In Window-LFU algorithm replacement decisions are made based on statistics for a subset of all objects that have been accessed in the past. [59].

There are certain disadvantages of this strategy. Using this strategy cache pollution can occur. Cache pollution is a phenomenon where the data inserted into the cache will not be reused before it is expelled. Another problem is due to the similar values of frequency. Many objects can have the same frequency count and a tie breaker is needed for selection.

LRU: LFU policy does not provide a solution where the characteristics of temporal and spatial locality exist in the data. LRU cache replacement policy is based on the temporal locality principle which states if some data is requested, then there is a high probability that it will be requested again in the near future. LRU is widely used in database and Web-based applications. Least recently used removes the object from the cache that was requested least recently. LRU policy works well with applications that show high locality of reference. All objects with higher probability of being requested in the near future will be kept in the cache. In Squid, A proxy server that filters Web traffic and caches frequently accessed files, the LRU is successfully used along with certain parameters to control the usage of the cache. LRU is also been used with several variations. LRU-Min favors smaller objects and expels the least recently used object with size at least S. LRU-Threshold only caches the objects that are smaller than a certain size [1]. In Hyper-G cache replacement algorithm, ties are broken based on recentness of earlier access and also by measuring the size of objects [137]. HLRU algorithm utilizes the history of caches objects and keeps the objects with the maximum hist value based on OldTimeofAccess and NewTimeofAccess [134].

One of the disadvantages of the LRU is that it only considers the time of the last reference and does not use the frequency of objects to evict the objects from the cache. This factor is very important for static websites. Another problem is this method doesn't combine recency and size in a useful, balanced way [103].

Hybrid: Many researchers have studied several other strategies. These strategies use LRU, LFU with size, cost functions or combination of several techniques to expel objects from the cache. The algorithm SIZE replaces the object having the largest size. LRU-Min keeps smaller objects in the cache and replaces least recently used objects which are above pre-defined threshold [98]. Cost based algorithms make the decision of replacement based on parameters related to time such as time of expiration, time of insertion in the cache and time of last access. Greedy-Dual-Size cache algorithm incorporates locality along with size and cost [21]. Generational Replacement algorithm store objects in lists. Each list i ¡ n contains objects that were requested i times. List n contains all objects with n or more requests. A request to an object causes its deletion in its current list and its insertion in the next list [99]. HYPER-G strategy combines LRU, LFU, and SIZE [137]. Although these strategies work well however, due to composite procedures, most of these strategies are more complex than LFU and LRU.

2.2.5 Caching Paradigms

Cache stores copies of content passing through it so that subsequent requests may be satisfied from the cache if certain conditions are met. The main objective of any cache is to correctly answer as many subsequent requests as possible, also called as hit rate of a cache. To accomplish this objective several cache paradigms have been used. These paradigms can be differentiated by the type of content and management of that content in the cache. Mainly the content is either static or dynamic and there are three ways to manage it. The cache can be managed as a passive cache (cold cache) or as an active cache (hot

cache). Along with these methods, prefetching techniques can also be used to administer the cache. Bellow is the complete description of each paradigm.

Passive Caching: A passive cache stores some content and returns the same content upon a latter request. For example, if clicking on a given URL always generates the same result page, the subsequent request adds no information from the user, cache can store this result page and subsequent request can be answered from the cache. Similarly, caching a query and its answer can be used to answer subsequent requests for the same query. Passive caching serves to reduce network traffic as well as latency and server load. All three types of caches, browser, proxy & server, can be implemented using passive caching approach. It helps to reduce network traffic, server load as well as access latency.

Caching documents passively to reduce access latency is extensively studied. As discussed earlier both zipfs law and locality of reference suggest that users tend to seek similar information over a period of time. Keeping this information in the cache can help answering subsequent queries. Each query answered from the cache results in a cache hit and moderate cache hit rate can be achieved using passive caching.

Passive caching implemented using the browser cache works for the users independently. Browser caches the pages on the local machine and returns it upon hit. Passive caching in this case works very well. This cache is useful when a client hits the 'back' button to go to a page they've already seen. Also, if same navigation images are used throughout any website, browser cache keeps a copy of it and sends it immediately on subsequent pages. Passive caching in this case reduces access latency significantly in case of a cache hit because the redundant user requests will all be kept in the browser cache and it reduces the access latency, server load and network traffic.

Passive caching implemented as proxies between users and web servers reduces server load, network bandwidth usage as well as user access latency [37][85][111]. Browser caching also uses passive caching approach and yield performance improvement. Abrams

[2] observed that a cache hit rate of up to 50% is obtainable by employing a proxy cache. The cache hit rate of an unlimited sized cache grows like a log, zipf's distribution, as a function of users of the proxy and number of request. This property was observed in Digital Equipment Corporation's proxy traces and University of California Berkley's proxy traces [21][43]. Similarly, Duska et al. observed this property in a number of traces from university proxies and ISP proxies [35].

Passive caching can also be deployed on the server side for performance improvement as users tend to seek similar information and send same queries to be processed by web servers. Traces of several web servers show the presence of repeated queries. In the Vivisimo trace, over 32% of the queries were repeated ones that have been submitted before by either the same user or a different user. In the Excite trace, more than 42% of the queries were repeated queries [139]. The study of very large log of AltaVista Queries shows that the average frequency of the queries in their trace was 3.97 [127]. Using the traces from Exite search engine, Markatos found that a large number of queries that are accessed several times and are excellent candidates for caching [86]. Xie et al. studied two real search engine traces and analysis yielded that about 30% to 40% of queries were repeated this repetition follows a Zipf distribution. Also these repeated queries were requested by different users hence, are good candidates for a server side cache [139]. By using server side caching these repeated queries can be answered more efficiently and access latency can be reduced.

Passive caching can improve the performance up to a certain level. In one study passive caching was able to reduce latency from 22%-26% [66]. Another study showed that the search engine query results may reach a hit rate up to 25% using passive caching [86]. However, this performance can be further improved by employing other caching methods. These methods are discussed next.

Active Caching: Active caching extends the performance of passive cache so that it can not only service requests that exactly match previous requests, but it can also fulfill requests

that can be answered by processing results of previous requests. This type of caching was termed as active caching, because the cache functions in a limited query processing role [83].

Increasing number of sources of information on the web is becoming dynamic, generated on-the-fly in response to a user requests. Personalized web pages, targeted advertisements, e-commerce applications, recommender systems etc. all are examples of dynamic applications. Passive caching work well with static content however, in dynamic environments the performance can be further improved by utilizing more advance approaches. Recent work has targeted extending the passive caching concept by storing the result of dynamic web requests and utilizing this content. Active query caching for database web servers has been considered as feasible and promising approach [81]

In dynamic environments, passive approach caches queries and their answers, and can fulfill request for an exact query match. This approach give performance gain however, it is possible to do more by using active caching approach. This can be accomplished, if the cache itself has a query processing capability, that is if cache is able to answer cached queries as well as those whose answers can be effectively estimated from the cache.

Most of the available work in the area of active caching is targeted toward semantic caching, one variation of active caching where query semantics are used. In the area of active caching Pei et al. proposed an Active Cache protocol to support caching of dynamic documents on the Web and through prototype implementation and performance measurements, show that Active Cache is a feasible scheme that can result in significant network bandwidth savings at the expense of moderate CPU costs [22]. In another study Luo et al. implemented an active caching approach in which server sends a query applet to the cache that implements a simple query processor. Test results show that the active query caching can achieve higher hit rates than passive query caching [82]. Levy-Abegnoli et al. proposed an active caching approach to support caching of dynamic contents. This

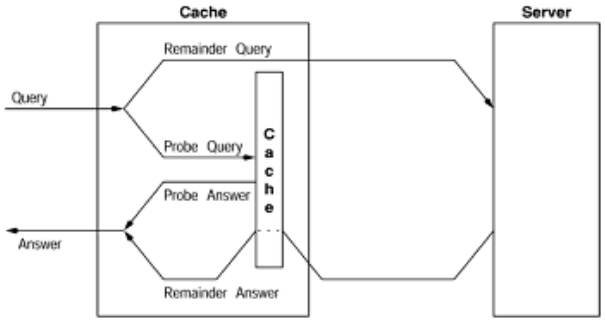

Figure 2.11 Advantages / disadvantages comparison chart.

approach provides an API which allows applications to add, delete and update cache data [73].

Semantic caching is a variant of active caching in which query semantics are utilized. These semantics are defined by inspecting a query predicates and operators used in this query. Semantic caching concept was first proposed by Dar et al. and they used the semantics of the queries to manage the contents of the cache and to decide about the availability or lack of query results in the cache [33]. Semantic caching manages the cache as a collection of semantic regions; these regions are formed by keeping the overlapping queries together and the size and shape of regions can change dynamically as new queries arrive in the cache. Access information is maintained and cache replacement is performed at the unit of semantic regions only [33]. In semantic caching, set of semantically associated results are group together in semantic regions as compare to tuples or pages which are used in conventional caching. When user sends a query, it is divided into two parts: a probe query that fetches the relevant portion of the answer set from the cache; and a remainder query that fetches the missing part of the result. If the remainder query is not empty, the remainder query is sent to the server for further processing [26]. In semantic caching the client is able to reason from the local cache to determine whether a query can be totally answered, how much it can be answered, and what data are missing. Semantic caching can improve performance substantially when a series of semantically associated queries

are asked and if the results may likely intersect or contain one another. In containment case where one query is a subset of another query, the answer is quite easily obtained from the cache without sending any remainder query. In overlap or intersection case however, the overlapping result is fetched from the cache and remainder query is formed to get the remaining result from the server as shown in Figure 2.11. This is an effective way to reduce costs by caching the results of prior queries and reusing them fully or partially to answer other queries [5][56]. Semantic caching can work well in applications, such as the cooperative database system and geographical information system [28]. Recently, semantic caching in a client-server or multi-database architecture has received attention [8][33][106][27] . Moreover, semantic caching is particularly attractive for use in mobile computing due to the fact of more autonomy of the mobile clients [71][106][145]. In [26], a semantic cache mechanism for Web queries based on signature files has been proposed. The method uses signature-based region descriptions to efficiently manage both containment and intersection cases.

Semantic caching works well with databases, mobile computing and in number of other areas. It has also been implemented in Web caching [26] [71]. Chidlovskii et al. used this method for conjunctive Web queries. A conjunctive query allows the Boolean operators AND and NOT between query terms. However, they implemented this approach without the operator OR, due to the exponential complexity of the semantic containment and intersection problem for the full Boolean expressions. Loukopoulos et al. proposed an active semantic caching approach that enables the proxies to cache some parts of the data, together with the semantics in order to process queries and construct the resulting pages [78]. Luo et al. proposed a method that uses this approach and suggested that computing answers for contained queries can provide significant improvement however, getting results for cache-intersecting queries is a challenge [83].

As discussed above, semantic caching in web environment is limited to only certain types of web queries. It cannot provide a solution for queries with OR operator. Also

the schemes mentioned above cannot be used for recommender system queries where queries are not conjunctive and there is no overlap between queries however, the result of the queries does intersect with each other. In this scenario application of these semantic caching techniques is not feasible and solution must be investigated by utilizing the results of the queries rather than semantics. This issue is discussed in detail in the next chapter.

In search engine domain, several studies have focused on caching approaches where cache acts in a query processing role. Saraiva et al. proposed storing inverted lists of query terms (keywords) in the cache to assist in the active caching query-handling process [116]. The cache mechanism uses a two-level scheme that combines cached query results and cached inverted lists. Results of repeated identical queries are cached at the front end, whereas data for frequently-used query terms are cached at a lower level. The inverted lists for each term are accessed, and used to generate lists of result documents containing all terms. Although their combined caching strategy increased the throughput of the system, it can only handle queries having multiple key- words. This approach can improve the performance in keyword queries however, it cannot provide performance gain in case of K-NN queries. K-NN queries are always based on a single object and their approach can only process already cached queries (passive caching). In [77] a three level cache was proposed wherein an intermediate level is added to the design. The intermediate level exploits frequently occurring pairs of terms by caching intersections or projections of the corresponding inverted lists. This approach also cannot accommodate K-NN queries, as it requires that the query be expressed as a pair of keywords which is not the case in K-NN queries. Luo & Naughton proposed an effective caching scheme that reduces the computing and I/O requirements of a Web search engine without altering its ranking characteristics [81]. Their approach targeted to answer the queries which are contained in already cached queries and mentioned that computing answers for contained queries can provide significant improvement however, getting results for cache-intersecting queries is a

challenge. Again query containment is not possible in K-NN queries and implementation of this approach can only give performance gain similar to a passive cache.

Although active caching provides a number of benefits over passive caching however, it has limitations or its own. CPU overhead, accuracy, and cache space management can easily gate the scalability of these schemes. These issues must be addressed before implementing active caching solutions. This research provides a novel active caching approach for recommender systems. Details of this approach are discussed in the Chapter 4.

Cache Prefetching: Pre-fetching is a process which utilizes idle time to download or pre-fetch documents that the user might visit in the near future. A web page provides a set of pre-fetching links, and after rendering the page to the end user, cache server begins silently pre-fetching specified documents and stores them in its cache. When the user visits one of the pre-fetched documents, it can be served up quickly out of the server's cache. Thus, successful prefetching reduces the delay and reduces both server and network load. Prefetching objects in proxies has been explored for further performance improvement by using various methods [22][65].

Pre-fetching is a proactive caching scheme because data is cached before the appearance of any request to that information. The pre-fetched information could be simply a static web page or other types of data, in case of dynamic websites, to service users' future requests. The main difference in efficiency as compared with the cache-only is in the first access of pre-fetched information. After being pre-fetched first time, it becomes part of cached data. In Pre-fetching, the important issue is how to predict the next information to be visited by the user. For static websites, within a page, it may have a number of combinations for identifying the next page to be accessed by a user. For dynamic pages, it is the prediction of future queries, for example, using usage statistics, and data required to service these queries. If the prediction is incorrect, the backend server will waste resources

on generating unwanted pages. However, due to the dynamic organization of web pages, it is not easy to have a high accuracy in prediction. Web prefetching technique utilizes the spatial locality of Web objects whereas, caching uses temporal locality. A page could have links to many other pages and all these pages are candidates for pre-fetching. Right prediction is very important in case of prefetching otherwise the overhead on the server can reduce the overall performance.

The pre-fetching problem is also complicated by the timing problem in pre-fetching. A pre-fetched page is useful only if it is available at the right time even if the prediction is correct. If the pre-fetched page is replaced before it is viewed then this entire effort will waste resources of backend server on generating unvisited information. Similar to prediction problem, browsing patterns information could be very useful in approximating the right timing of pages. Web caching and Web prefetching can complement each other since the Web caching technique exploits the temporal locality, whereas Web prefetching technique utilizes the spatial locality of Web objects [131]. Prefetching is a well-known approach to decrease access times in the memory hierarchy of modern computer architectures [128][126][79] and has been proposed by many as a mechanism for the same in the World Wide Web [100][65]. Marc et al. stated that maximum hit rate obtainable by using proxy cache is about 50% however, prefetching can be used to further the performance [2]. Padmanabhan and Mogul showed that prefetching can reduce the access latency experienced by the users by a maximum of about 45% [100]. Kroger et al. noticed that proxy caching can reduce the latency by maximum of about 26% whereas a combined caching and prefetching proxy yield 60% latency reduction [65]. Markatos and Chronaki used a top 10 approach to prefetching and their results showed that this approach can predict more than 40% of the client's requests [87].

Although prefetching provides performance benefits over simple caching techniques however, flawed use of this technique can undermine the benefits. Bad prediction and timing problem are among the reasons that can increase the network traffic and server load.

2.2.6 Summary

Caching of data has shown to be useful for reducing stress on limited resources and improving response time. Selection of cache location, browser, proxy or server cache, primarily depends on the type of application. Cache replacement policies help in maximizing the hit rate of cache and selection of best method depends upon the actual usage pattern of the system. Selection of paradigm is a tradeoff between effectiveness and efficiency and should be selected based on the sensitivity of the application.

CHAPTER 3

CACHING FOR RECOMMENDER SYSTEMS

Research literature in the area of recommender system has very limited to no discussion on caching solutions for these applications. Recommender system queries or top-k similarity queries (also known as k-nearest-neighbor, or k-NN queries), the result of a query is a ranked list of items based on relevancy score. Generating these ranked lists is typically an expensive operation and often results in access latency. Traditional caching solutions can easily be used with any type of recommender system. Traditional cache management strategies generally seek to fill the cache with result lists for the most popular queries, and to utilize effective replacement strategies to maximize the overall performance. In general, only limited performance gains are possible with this type of caching. This dissertation focuses on exploring a caching solution for recommender systems that improves upon the performance of traditional caching. This chapter first examines the characteristics of recommender system queries, then explores the possible caching options available in other research areas and finally, formulates the research questions for this dissertation. It also provides the description of different datasets along with the evaluation measures that will be used to evaluate the performance of the proposed solution.

3.1 Recommender System Queries

Recommender system queries in general are considered k nearest neighbor type queries (K-NN queries) or also termed as query-by-example type of queries . In these types of queries, the query is always composed of one object and the result is a ranked list of k nearest neighbors where k is a predefined value and nearest neighbor selection is based on the type of recommender system and matrix used.

Let S be a dataset drawn from some domain D. For every object $v \in S$, implies the existence of a unique ordering $(v_1, v_2, \ldots, v_{|S|})$ of the objects of S, where $i < j$ implies that v_i is deemed more relevant or similar to v than v_j. With respect to v, the rank of object w ranges from 1 to $|S|$, and will be denoted by *rank*(v, w). In practical settings, the object most relevant to v is generally v itself. Nevertheless, unless otherwise stated, it is not require that *rank*$(v, v) = 1$.

3.2 Caching Options

Based on the above definition of recommender system queries, a caching solution for these queries is discussed below. These are several caching specific alternatives that should be carefully selected for example, cache location, caching paradigm, and cache replacement policy as shown in Figure 3.1. First, the best location for a recommender system cache. Second, the selection of the best paradigm that can improve upon the performance of traditional caching and finally, appropriate cache replacement policy which works best with the selected paradigm.

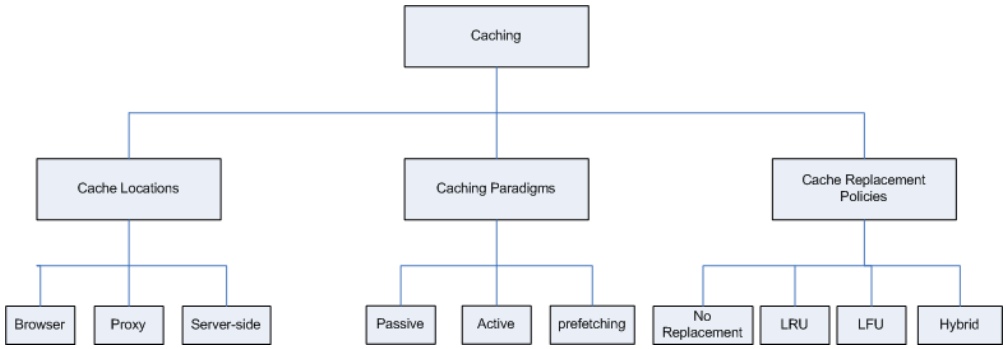

Figure 3.1 Available caching options for recommender systems.

3.2.1 Cache Location

There are three most common locations where caches are normally implemented. Browser caches are at the client side, proxy caches are places between servers and clients and server side caches are implemented on the server side. Both browser cache and proxy cache are best for static content, however, dynamic applications like recommenders systems use backend processing to compute ranked lists. Implementation of cache for such applications on the client side is out of question as each cache can only serve one client and when millions of people are accessing the application only duplicate queries from a single person will get benefit from cache. Proxy cache solutions are most viable for intranets where proxy can cache the request from a small community and the content is mostly static. Applications like recommender systems that serve the large user base around the globe and the content is dynamic, recommendation list is generated on the fly, best possible location is server side cache. Server side cache for recommender system will keep those queries and their result in the cache that have higher chance of being requested latter. Each request coming to the recommender system server will be first checked in the cache. If the answer is available in the cache, cache hit occurs and answer will be sent back from the cache. If answer is not available in the cache then it will be requested from the recommender system and then sent to the user. Based on the cache replacement policy, a copy of this answer might be retained in the cache for latter requests for same query.

3.2.2 Caching Paradigm

The purpose of maintaining a cache is to allow faster delivery of query results to the user. Accordingly, answering a query with the aid of a cache should be faster than retrieving a result from the server and the query result obtained using the cache should be as similar as possible to the result that would be retrieved from the server.

Keeping in view these goals, cache for a recommender system can be implemented using any of the three paradigms discussed in the last chapter. Each paradigm has its own advantages and disadvantages as discussed below.

Passive Caching: When a server query result is duplicated in the cache, the technique is called passive caching. Passive caching approaches only attempt to answer those queries whose result is available in the cache and remaining query results are requested from the recommender system. Passive cache management strategies generally seek to fill the cache with result lists of popular queries or recent queries, and utilize effective replacement strategies to maximize overall performance.

Implementation of passive caching for recommender system is very straight forward similar to other applications like search engines. Cache will keep the answer for most popular or most recent queries, based on replacement policy, and only these queries can be answered from cache if requested latter. Any such technique will guarantee the perfect recall of the results answered from the cache as the answer was previously fetched from recommender system and provided to the user as is. However, using this approach can only provide limited performance benefit in terms of hit rate. Various cache replacement policies can help to increase the hit rate to a certain extent but with organizations having millions of users and millions of items it becomes a bottleneck. In these circumstances passive caching can only provide limited performance benefit.

As mentioned earlier, none of the prior studies discussed any caching solution for recommender systems. However, in other types of retrieval systems many studies have focused on this topic. One example of a retrieval system for which passive caching was effective is the Excite search engine. Markatos discovered a large number of frequently-posed queries in the retrieval logs that constitute excellent candidates for caching [86]. However, limited performance gain was possible with passive caching in a study showing that search engine query results may achieve a hit rate up to 25% using passive caching

[88]. In experiments involving Digital Equipment Corporation traces, passive caching was able to reduce latency by 22%-26% [66].

Although passive caching is a viable solution, scalable applications like recommender systems need higher performance improvements. Active caching and cache prefetching are two possible ways to achieve this goal.

Cache Prefetching: Pre-fetching is a proactive caching paradigm because data is being fetched into the cached before the appearance of any request to that information. Existing studies have showed that prefetching combined with passive caching can potentially improve the latency [25]. However, mostly this technique is used for static content like web pages etc. in which case images or text can be prefetched before the request from user.

Dynamic applications like recommender system where millions of users will be sending individual requests for recommendations pose a real challenge to prefetching techniques. Particularly for caches of small size, prefetching might negatively affect the effectiveness of the cache replacement policy adopted. In case of cache replacement the prefetched pages of results have to be inserted in the cache by likely evicting from it an equal number of entries according to the replacement policy adopted. Obviously, the hit rate increases only if the probability of accessing the prefetched pages is greater than the evicted ones. Moreover, this whole process of prefetching increases the load on the back-end server and increase network traffic which can overall degrade its throughput. Also making prediction of what data to prefetch is very crucial and difficult. In case of commercial recommender systems like Amazon.com it becomes even much more difficult with millions of users' requests have to be analyzed to see what next they might be seeking that need be prefetched.

From these limitations, it is clear that although prefetching is a viable solution but, not practical for recommender system cache. There is a dire need for a caching solution that should increase the hit rate but not at the cost of degradation to its throughput.

Active Caching: Active caching extends the performance of a passive cache so that it can not only service requests that exactly match previous requests, but it can also fulfill requests that can be answered by processing results of previous requests. This type of caching was termed as active caching, because the cache functions in a limited query processing role [83]. When a user sends a query there are three possibilities. First, the answer for that query is already available in the cache. Second, although the answer is not available in the cache but it can be computed from the cache. Third, the answer is neither available in the cache nor it can be processed so it is fetched from the recommender system. First case is simple passive cache where as third case is direct answer from the recommender systems. For the second case, active cache, where answer is estimated from the cache, domains like search engine and databases have some studies in this area.

The aim is to first explore available active caching solutions in similar domains and see if those can work for recommender systems. One example of an active caching strategy is semantic caching, which is based on the assumptions that the queries submitted to information retrieval systems are boolean, and that the results of previous queries can be used to compose results for new queries, using boolean algebra. When the user submits a query, it is decomposed into two parts: a probe query that fetches the relevant portion of the answer set from the cache, and a remainder query that fetches the missing part of the result. If the remainder query is not empty, it is sent to the server for further processing [26]. In the case of a containment query, when one query is a subset of other query, the final result is easily obtained from the cache without requiring the generation of a remainder query. However, in the case of an intersection query, the portion of the result in the intersection is fetched from the cache, and the server is queried to obtain the remainder of the result.

Semantic caching is particularly attractive for use in mobile computing platforms, due to the greater autonomy of the mobile clients [71][106][145]. It has also been used in web caching to handle conjunctive queries supporting the use of Boolean operators AND and NOT between query terms [26]. However, the operator OR could not be supported,

due to the exponential complexity of the semantic containment and intersection problem for full Boolean expressions. Luo et al. proposed a method that uses the semantic caching approach; they concluded that answering cache-contained queries results in a significant performance gain, but answering cache-intersecting queries is probably not worthwhile for top-k conjunctive keyword queries [83]. The main limitations of using semantic caching are that it can only work with boolean queries and utilize query containment. However in case of K-NN queries, query is always based on a single object hence, query containment is not possible. Any two queries are always based on different objects and are never contained in each other and even never intersect each other. Thus if implemented, use of semantic caching for recommender system queries can only work like a simply passive cache.

In search engine domain, several studies have focused on caching approaches where cache acts in a query processing role. Saraiva et al. proposed storing inverted lists of query terms (keywords) in the cache to assist in the active caching query-handling process [116]. The cache mechanism uses a two-level scheme that combines cached query results and cached inverted lists. Results of repeated identical queries are cached at the front end, whereas data for frequently-used query terms are cached at a lower level. The inverted lists for each term are accessed, and used to generate lists of result documents containing all terms. Although their combined caching strategy increased the throughput of the system, it can only handle queries having multiple keywords. This approach can improve the performance in keyword queries; however, it cannot provide performance gain in case of K-NN queries. K-NN queries are always based on a single object and their approach can only process already cached queries, a case of simple passive cache.

Xiaohui et al. proposed a three level cache wherein an intermediate level was added to the design [77]. The intermediate level exploits frequently occurring pairs of terms by caching intersections or projections of the corresponding inverted lists. This approach also cannot accommodate K-NN queries, as it requires the query to be expressed as a pair of keywords which is not the case in K-NN queries.

Luo & Naughton proposed an effective caching scheme that reduces the computing and I/O requirements of a Web search engine without altering its ranking characteristics [81]. Their approach targeted to answer the queries which are contained in already cached queries and mentioned that answering contained queries results in a significant performance gain, but answering cache-intersecting queries is probably not worthwhile. Again query containment is not possible in K-NN queries and implementation of this approach can only give performance gain similar to a passive cache.

From the above discussion it is clear that already available active caching strategies cannot provide performance gain in case of K-NN queries. If implemented, all these approaches can only provide answer for the queries which are already cached (passive caching) and cache itself cannot work in a query processing role. Most of the above mentioned active caching approaches use the query containment and query intersection to process answer from the cache. There is a need for an approach that does not use the part of queries answers but process the answers of the previously cached queries to answer non-cached query.

This study investigates an active caching solution for recommender systems. Active caching is an extension of the caching model whereby estimation is used to generate an answer for queries whose results are not explicitly cached, where the estimation makes use of the results cached for related queries. By answering non-cached queries along with cached queries, active caching approach offers substantial improvement over traditional caching methodologies. This dissertation focuses on the problem of active caching for recommender system or top-k similarity queries. The goal is to estimate an answer for a recommender system query using only cached information and without performing expensive disk access operations, the computational savings may be considerable.

3.2.3 Cache Replacement Policy

Normally only a small proportion of the entire dataset is cached, and the criteria by which cache items are selected is crucial to the performance of any caching solution. This selection can be performed in many ways; however, the ultimate goal of selection is to maximize the number of queries that can be answered from the cache.

Several techniques have been proposed in the research literature to select the most appropriate data for caching. Typically, the content of the cache is dynamically updated in order to adapt to changes in user request patterns. Insertion of new items into the cache first requires that items be selected for replacement. Most cache replacement strategies select for deletion either the least recently used cache element (LRU) or the least frequently used element (LFU). Both the LFU and LRU cache replacement strategies take into account the popularity of the data with respect to query requests. The LRU approach can be viewed as a form of temporal locality, whereas the LFU approach can be viewed as a form of zif's law in that it preserves cache objects residing in areas where the query distribution is dense.

Although traditional caching strategies allow for dynamic updates, researchers have also considered the problem of selecting a static cache so as to be able to answer the maximum number of queries for given distributions of data and queries [130]. Active caching approach utilizes the data in a cache to answer maximum number of queries. Hence, using a static cache with a better uniform coverage of the query range can increase the spatial locality from which most if not all query results can be actively generated.

3.2.4 Active Cache for Recommender Systems

Selection of best caching options for recommender systems from the above mentioned choices is obvious in some cases like cache location where best option is to implement a service side cache. However, decision about the paradigm selection need to be done by evaluating the pros and cons of each approach. This study opted to implement an active caching solution in order to achieve maximum performance gain. Active caching solution

can provide significantly higher hit rate but at the cost of lower recall which can be tolerated in non-critical applications like recommender systems. In case of cache replacement policy, a spatial locality based static caching policy is opted because it augments with an active caching approach. These decisions were made based on the research questions outlined in the next section.

3.3 Research Questions

This study is focused on providing a solution for the efficiency of recommender systems by caching appropriate content in the main memory and then answering most if not all of the queries from the memory rather than requesting them from the recommender system. To design such a system, three major research issues need to be investigated: First question is how to design a caching solution that can answer more queries than those available in the cache. This design should be robust enough to work with any type of recommender system and does not require access to the underline matrix. Also the solution should work in cases where the access patterns does not follow zipf's law or temporal locality. Active caches estimate answers for non-cached queries so the second question is how to maximize the result accuracy of non-cached queries which are estimated from the cache. Third research question that should be investigated is how and what type of data should be selected to put in the cache to achieve maximum performance from the cache. More specifically, the following main research questions are to be answered:

Questions No 1:

How to design an effective and efficient caching solution for recommender systems?

Question No 2:

How to design a more general and effective similarity measure for active caching?

Question No 3:

How to select the objects in the cache for a caching with no replacement?

3.4 Evaluation

Active caching technique proposed in this work is designed for recommender systems in specific however, it can be used with any nearest neighbor application in general. In Chapter 2, recommender systems are categorized into four major categories i.e., content based, collaborative filtering, knowledge based and hybrid systems. Most of the commercial recommendation systems use either content based or collaborative filtering approach to process the recommendations. Knowledge based systems have only been used in few research studies and appropriate datasets for these systems are not available. Hybrid system are being used by several commercial applications however, the techniques used for processing the recommendations are based on the combination of content based and collaborative filtering techniques. Each of these systems, content based and collaborative filtering, can be implemented in several ways, however, as mentioned in Chapter 2 these can be categorized by the type of matrix used. Matrices are based on the distance function used and are usually characterized by metric and non-metric distance functions. To cover both content based and collaborative filtering recommendation techniques, five different types of datasets and both metric and non-metric distance functions are being used to evaluate the performance of proposed active caching approach using hit rate, byte hit rate, recall and execution cost measures.

3.4.1 Dataset

Content Based Datasets: Content based systems use the content of each object in the collection to compute recommendations based on the similarity amongst the objects in the collection. Content based systems are usually used for collections where content can easily be extracted to measure the similarity amongst the objects. Type of matrix used to compute the similarity is another way to categorize content based systems which are either metric or non-metric. Most commonly used matrices are based on Euclidean distance, cosine similarity and Pearson correlation where Euclidean distance is a metric distance measure

and both cosine similarity and Pearson correlation are non-metric measures. In order to make the test datasets diverse enough two different types of object collections are chosen, a text document collection or digital library collection and an image collection, as well as both metric and non-metric matrices, Euclidean distance and cosine similarity, are used to compute similarity. Each dataset is described in detail below

Reuters Dataset of Digital Documents: Reuters Corpus Volume I (RCV1) is an archive of 802,352 newswire stories made available by Reuters, Ltd. for research purposes [74]. Reuters is the largest international text and television news agency. Its editorial division produces some 11,000 stories a day in 23 languages. Stories are both distributed in real time and made available via online databases and other archival products. The set consisted of 802352 documents and are modeled as bags of words. The feature words are extracted and weighted to formulate a vector of words for each document, TF-IDF term weighting [114], and their 120-NN lists were constructed using the SASH approximate similarity search structure [50] lists with the vector angle distance measure. The degree of relevance is measured by the similarities between the vectors of words for these documents. Cosine similarity, a non-metric distance measure, is used to calculate the degree of similarity amongst the documents. In this case, two items are thought of as two vectors in the m-dimensional user-space, where dimensions are correspond to the words in each document. The similarity between them is measured by computing the cosine of the angle between these two vectors. Formally, in the "mxn" ratings matrix similarity between documents A and B, denoted by $sim(A, B)$ is given by

$$\mathbf{CM}(A, B) = \frac{|A \cap B|}{\sqrt{|A| \cdot |B|}}$$

The similarity calculation process results in a relation with $n2$ number of tuples where n are the total number of documents however, as only top 120-NN were stored for each object. Recommender system uses this relation tuples to answer the queries posed by the users.

Given a query, the nearest k (k is a predefined parameter) documents in the collection are found and presented to the user as the similar documents.

ALOI Image Collection: ALOI dataset is an image collection based on Amsterdam Library of Object Images (ALOI) [39]. The full dataset consists of 110250 images of 1000 common objects taken from a number of different angles under different lighting conditions. The images were represented by dense 641-dimensional feature vectors based on color and texture histograms [16] and 100-neighbor lists were computed using the Euclidean distance for each image of the collection. Euclidean distance is based on distance between objects as compared to similarity between them. In similarity measures, higher the similarity more similar are the objects where as in distance measure, lower the distance more similar are the objects. Here similarity and distance are used interchangeably. Euclidean distance examines the root of square differences between coordinates of a pair of objects and is a straight line distance between two points. If $a = (x_1, x_2, \ldots, x_n)$ and $b = (y_1, y_2, \ldots, y_n)$ are two points on the plane, their Euclidean distance is given by

$$d(a, b) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}.$$

Geometrically, it's the length of the segment joining u and v, and also the norm of the difference vector (considering $\mathbb{R}^n$ as vector space). This distance induces a metric (and therefore a topology) on $\mathbb{R}^2$, called Euclidean metric (on $\mathbb{R}^2$)and standard metric on $\mathbb{R}^2$.

Cover Type Geographic Data: This dataset is prepared for forest cover type from cartographic variables only (no remotely sensed data). The actual forest cover type for a given observation (30 x 30 meter cell) was determined from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. Independent variables were derived from data originally obtained from US Geological Survey (USGS) and USFS data. Data is in raw form (not scaled) and contains binary (0 or 1) columns of data for qualitative independent variables (wilderness areas and soil types). Number of instances (observations)

in the data are 581,012 where as each observation has 12 measures, but 54 columns of data (10 quantitative variables, 4 binary wilderness areas and 40 binary soil type variables). 100-NN lists were constructed using the SASH approximate similarity search structure using Euclidean distance [50]

KDD Cup Intrusion Detection Data: This is a standard set of data, which includes a wide variety of intrusions simulated in a military network environment provided to the participants of The Third International Knowledge Discovery and Data Mining Tools Competition. The datasets contain a total of 24 training attack types, with an additional 14 types in the test data only. A small subset of 120836 observations were used for this study. This set represents a content based system and Euclidean distance is used to compute the distances amongst the objects. 100-NN lists were constructed using the SASH approximate similarity search structure [50]

Collaborative Filtering: Collaborative filtering methos do not take into consideration the content of the objects rather use different data gathering techniques to process the recommendations. Number of data gathering techniques like rating information, browsing history, purchase history etc. as well as number of CF algorithms e.g., user based, item based, aggregation etc. have been used to process recommendations. CF algorithms use one of the distance functions mentioned earlier to predict the likelyhood of an object and these functions are categorized into metric and non-metric types. Due to the limited availablility of the datasets, only one underneath datasource was used. However, to make it diverse, two different types of CF algorithms, user based and count model were used.

MovieLens Dataset: MovieLens data sets were collected by the GroupLens Research Project at the University of Minnesota [46]. This data set consists of 100,000 ratings (1-5) from 943 users on 1682 movies. Each user has rated at least 20 movies. The data was collected through the MovieLens web site (movielens.umn.edu) during the seven-month period from September 19th, 1997 through April 22nd, 1998. This dataset was used to

test active caching for collaborative filtering systems. Model based approach was opted to generate recommendations and used Multilens [93] to generate these models. MultiLens is an OpenSource Recommendation Engine and supports several kinds of recommendation algorithms including user-to-user correlation, item-to-item correlation and count model (represent similarity by simple co-occurrence). MultiLens is used to make recommendations on the MovieLens movie recommendation web site, and is being used in Other research projects by the GroupLens group at the University of Minnesota. This study uses Count model approach to process the recommendations. The basic functionality of a CountModel is to keep track of the number of times an item has co-occurred with another. Incrementing the co-occurrence counter is a fundamental operation for this model. Given a user id and a new item rated by that user, system incorporates this new item into the model along with all of the previously rated items by this user.

Jester Rating Dataset: This dataset represents a collaborative filtering recommendation system, which used data like rating information, browsing history, purchase history etc. rather than content of the objects itself. Anonymous ratings data from the jester online joke recommender system is used. It is Collaborative Filtering Data of 4.1 Million continuous ratings (-10.00 to +10.00) of 100 jokes from 73,421 users collected between April 1999 - May 2003. Euclidean distance measure was used to compute the similarity amongst users using the available rating data.

3.4.2 Performance Measures

In web caching research cache hit rate is the most common measure for evaluation. In passive caching answer for a query is readily available in the cache hence, precision and recall is always perfect i.e., 1. However, in active caching, recall and precision for the cached queries is always perfect but for non-cached queries which are processed from the cache these values vary. In order to do the evaluation for effectiveness, recall measure is used in this study. When the cache query and the database query have equal size k, this

definition is equivalent to that of query precision. Execution cost is also used as another measure for efficiency of the proposed solution. Each of these measures are explained in detail below:

Hit Rate: The hit rate is defined to be the proportion of queries that lead to cache hits. Traditionally, a hit is said to occur when the information sought for resides in the cache. In active caching, this definition is extended to include those cases also where a query result can be processed using the proposed methods. A miss occurs when a query cannot be processed from the cache and the answer is fetched from the recommender system.

Byte Hit Rate: The byte hit rate is is the hit rate with respect to the total number of bytes in the cache that lead to cache hits. Similar to hit rate, this definition includes cases where query result is available in the cache as well as those cases where a query result can be processed using the proposed methods. A miss occurs when a query cannot be processed from the cache and answer is fetched from the recommender system.

Recall: Consider the item set retrieved by any given top-k query operating on the cache. The recall of the query is defined as the proportion of this result that would also appear in a top-k query applied to the full database. Since the cache query and the database query have equal size k, this definition is equivalent to that of query precision. Note that when the top-k list is explicitly stored in the cache, the recall trivially equals 1, and when a cache miss occurs, the recall is 0.

Execution Cost: Execution cost is the time a systems takes to process certain number of queries. In this study execution cost is measured in terms of time to process all the queries in the dataset. These queries could be those whose answer is readily available in the cache, or whose answer can be estimated from the cache or the ones whose answer cannot be provided from the cache and is fetched from the database.

3.4.3 Hardware and Software

Microsoft SQL Server is used to store the recommender system matrix in order to process recommendations. All the implementation was done in Microsoft CSharp and tested on an IBM machine with processor speed of 3.0 GHz and memory of 2 GB.

3.5 Summary

Caching solution for recommender systems can be implemented in different ways. However, for maximum performance, active caching is a viable approach. Active caching allows to answer significantly higher number of queries from the cache with reasonable recall rates. Applications like recommender systems, which are not mission critical in terms of recall and precision, can attain significant performance gain through active caching. This study has focused to provide an active caching solution including cache management policy and data structure which can be used with any type of recommender system to increase its efficiency. In the next few chapters this active caching approach is presented and evaluated using the above mentioned datasets. Chapter 4 introduces the partial order based active caching technique for recommender system. Chapter 5 proposes an improvement to the original approach by introducing shared neighbor similarity measure which helps to improve the recall. Chapter 6 introduces a greedy balancing cache selection policy which helps to improve the hit rate and recall of the proposed active caching approach. Finally, Chapter 7 highlights the contributions of this work and future research directions.

CHAPTER 4

PARTIAL ORDER BASED ACTIVE CACHING APPROACH

4.1 Introduction

As discussed in the previous section, caching has enabled developers to optimize the performance of web applications. Cache keeps a local copy of data to reduce visits to the actual data storage location. Cache size is always much smaller than actual storage area and choosing the right objects to keep in the cache is very important. However, answering already cached queries can only provide limited performance benefit. This chapter proposes a new active caching based solution that can not only answer already cached queries but can also provide answer for queries other than those in the cache

The proposed active caching technique is based on partial orders that can be used with any type of recommender system. It uses the principle of monotonicity of rank order to construct results for non-cached queries. This approach can not only answer queries that are already cached, but is also capable of efficiently synthesizing answers for 'neighboring' queries using the contents of the cache. In case of recommender systems, neighboring queries are all those objects that exists in the result list of a top-k query and this query happens to be in the cache. These neighborhood queries have higher chance of being accessed in the future as laid out by spatial locality principle.

The proposed technique is primarily targeted for recommender systems and works as a server-side cache. This work has investigated the question of whether a caching strategy might be useful as optimization technique for recommender systems. Answering a query in recommender systems requires a significant amount of computation and resources. Benefits of caching popular queries and their answers could be two fold. Firstly, repeated queries could be answered without redundant processing to minimize the access latency. Secondly, because of the reduction in server workload, resources could be used for other queries to

be answered more efficiently. Although Web caching has been widely studied, no explicit work is available in the domain of recommender systems.

Traditional caching implementation is very straight forward for recommender systems however, aim of this study is to investigate a technique that can not only answer queries that are already cached "passive caching" but can also actively process, "active caching", other queries utilizing information available inside the cache. As discussed in the earlier chapter, passive caching approach caches a page and returns it on a hit without any extra processing. Cache hit rate can be further improved by using active caching approach. The active cache can not only service requests that exactly match previous requests but also service requests that can be answered by processing results of previous requests [81]. The goal is to not only get benefit from temporal locality but also focus on spatial locality and process neighboring queries by using the cached information.

4.2 Partial Order Based Active Caching Solution

4.2.1 Preliminaries

Let S be a dataset drawn from some domain D. For every object $v \in S$, the existence of a unique ordering $(v_1, v_2, \ldots, v_{|S|})$ of the objects of S, where $i < j$ implies that v_i is deemed more relevant or similar to v than v_j.Assume query Q represents a recommender system query and its results is a ranked list l. In other words l=$Q(v, k)$, a nearest-neighbor type of query is sent to a recommender system which generates the list l = $(v_1, v_2, v_3, \ldots, v_k)$ of the *top k* objects that are the closest to the query object v. According to the temporal locality principle, query $Q(v, k)$ is likely to be asked again in the near future and should be kept in the cache. Lets assume C is a main memory cache and $C \subseteq S$. C is used to keep $|C|$ most repeated queries in the cache.

Traditional caching approaced target to keep repeated queries in C and can only answer subsequent requests for these queries. In applications like recommender systems requests are less likely to be repeated hence, minimizing the benefits of caching. This

study proposes an active caching approach whose goal is to provide answers for not only cached queries but also non-cached queries. Spatial locality suggests that queries that are close (similar) have a higher probability to be asked in the near future and this approach targets to answer these similar non-cached queries. Generally in a Nearest-Neighbor type of queries, every object is also a query. In this case $(v_1, v_2, v_3, \ldots, v_k)$ can also be possible queries with a high probability in the future since being close to or similar to Object v as per spatial locality principle. If a solution is able to accurately answer $(Q(v_1, j_1), (Q(v_2, j_2), Q(v_3, j_3), \ldots, (Q(v_k, j_k))$, $j_i \leq k$ from the data in the cache, it will also achieve a much higher hit rate as the cache will also be working as query processor for some queries.

4.2.2 Cache Implementation

Consider now the situation in which a main-memory cache $\mathcal{C}(C, k) \triangleq \{\mathbf{Q}(v, k) : v \in C\}$ of top-k relevant sets is available for each object in a given subset $C \subseteq S$, for some fixed $k \in [1, |S|]$. If each of the relevant sets is maintained as a list of objects sorted from most relevant to least relevant, the collection of relevant sets $\mathcal{C}(C, j)$ is also readily available for all $1 \leq j < k$. It is referred to $\mathcal{C}(C, j)$ as a *sub-cache* of $\mathcal{C}(C, k)$. The *support* of an object v in the cache is the number of list in which v appears and is denoted by $Supp(v)$.

For a given $u \in S$, reverse relevant sets for u can also be defined with respect to the cache $\mathcal{C}(C, k)$, by restricting the membership of the lists to objects of C instead of S, as follows:

$$\mathbf{Q}_C^{-1}(u, k) \triangleq \mathbf{Q}^{-1}(u, k) \cap C = \{v \in C : u \in \mathbf{Q}(v, k)\}.$$

The collection of all such reverse lists taken over all choices of $u \in S$ will be referred to as the *reverse cache* corresponding to $\mathcal{C}(C, k)$, and will be denoted by $\mathcal{C}^{-1}(C, k) \triangleq \{\mathbf{Q}_C^{-1}(v, k) : v \in C\}$. In this work the terms *forward cache* and *forward relevant set* are used to refer to the original cache $\mathcal{C}(C, k)$ and its lists, and the term *cache* loosely to refer

to the set C taken together with its forward and reverse relevant sets. Figure 4.1 shows a structure with set size of 10, cache size of 3 and reverse lists updated according to their presence in the forward lists. All the objects with forward relevant set available in the cache can be answered as is, same as traditional caching approach. For the object without forward revelent set in the cache, inverse relevant set will be used to process the answer from the cache and cache will act in a limited query processing role or referred as active cache.

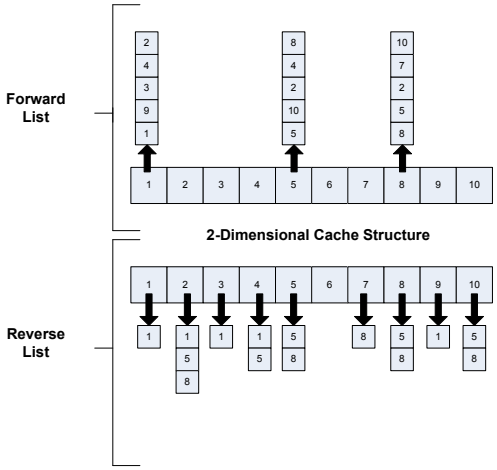

Figure 4.1 Cache data structures for a set of 10 objects in the 2-D plane. Top-5 lists are cached for three objects and reverse lists updated accordingly.

4.2.3 Partial-Order Based Approach

The proposed method for active caching of recommender system results is described in detail below. The method does not make use of the actual distance values, instead rely only on ranking information of query result lists. This approach uses partial ordered list characteristics to compute the answer for neighboring queries. In order to assess the impact of using rank information instead of distance information for active caching, study also describes and implements a distance-based variant of this method.

Let S be a dataset drawn from some domain D and l is a ranked list of objects similar to the query object returned by a recommender system for S. Let *rank*(l, v) denote the rank of the object v in the list l. A preference (or dominance) relation over l is a binary relation $\succ$ over $l \times l$, where $v_1 \succ v_2$ whenever *rank*$(l, v_1) >$ *rank*(l, v_2). In such situations, v_1

is *preferable* to v_2, or that v_1 *dominates* v_2. When extended to the full domain D, the preference relation constitutes a partial order on D. If for $v_1, v_2 \in D$ neither $v_1 \succ v_2$ nor $v_2 \succ v_1$ hold, then v_1 and v_2 are incomparable, written $v_1 \sim v_2$.

Given a query object v for which no result is cached, The active cache method first generates two partial orderings from each cached query result list containing v as shown in Figure 4.2:

- the *suffix list suff*(l), defined as the sublist of l consisting of items with ranks strictly higher than *rank*(l, v); and
- the *prefix list pref*(l), defined as the sublist of l consisting of items with ranks strictly less than *rank*(l, v), taken in reverse order.

With respect to these two partial orderings, define the rank *rank*$_v(l, u)$ of object $u \in l$ with respect to v to be the rank that u holds in either *pref*(l) or *suff*(l) — note that u cannot simultaneously be contained in both. More precisely, this rank is defined to be the difference

$$\textit{rank}_v(l, u) \stackrel{\triangle}{=} |\textit{rank}(l, u) - \textit{rank}(l, v)|.$$

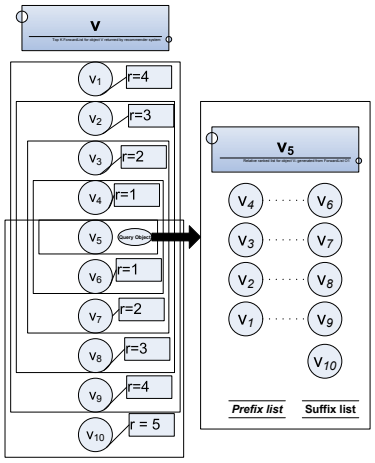

Figure 4.2 Structure showing the example extraction of prefix and suffix lists for object v_5.

The objective of active caching is to synthesize an answer for a given query from other query result lists available in the cache. Assume l is a cached list with $\textit{rank}_v(l, v_1) > \textit{rank}_v(l, v_2)$, for some pair of objects $v_1, v_2 \in D$. If for a top-k query-by-example $Q(v, k)$ an active cache method generates a ranked result list l' containing both v_1 and v_2, then ideally it would expect the ranks of these objects in the synthesized result to satisfy $\textit{rank}(l', v_1) > \textit{rank}(l', v_2)$. In this limited sense, the active cache function would be 'monotonic' with respect to l, in that the ordering of the objects v_1 and v_2 would be preserved. However, note that it is in general impossible for any aggregation function to be monotonic with respect to all cached lists. Monotonicity applies only to the lists selected from the cache to contribute to the answer and is only used in the aggregation algorithm to determine the ranks of the object v_1 and v_2. For recommendation systems, monotonicity makes sense because most objects in a list will appear in each others lists. The active caching method proposed in this dissertation resolves conflicts in the partial order information by aggregating the rank information across all suffix lists and prefix lists available from the cache. The aggregation can be performed with respect to such standard operations as *min*, *max*, *avg* and *skyline*.

Algorithm Query

Input: query object v, result size k;

Output: ranked top-k query-by-example result list *Result*.

1. Initialization:
   (a) Assign list $L \leftarrow \mathrm{Q}_C^{-1}(v)$. L refers to the cached forward lists containing object v.
   (b) Initialize result object candidate set *candset* $\leftarrow \emptyset$, and final result object set *Result* $\leftarrow \emptyset$.
2. For all lists $l \in L$ do:

(a) For all objects $u \in l$ do:

i. If $u \notin$ *candset* then insert u into *candset*, and initialize rank value multiset *rankset*$(u) \leftarrow \emptyset$.

ii. Insert an instance of the rank value *rank*$_v(l, u)$ into the multiset *rankset*(u).

3. For all objects $u \in$ *candset* do:

(a) Generate a score s for u by aggregating the rank values stored in *rankset*(u) using the chosen aggregation function (for example, *max*, *min*, *avg*, etc.).

(b) Insert the object-score pair (u, s) into the result list *Result*.

4. Sort the entries in the list *Result* according to their score values, and return the objects of top k object-score pairs. Ties can be broken arbitrarily, with the exception that v is given priority over any other object $w \neq v$ in D.

If the object domain D is embeddable in a metric space M with distance metric d, and these distance values are readily computable, a distance-based variant of the proposed query algorithm is possible: each instance of the rank *rank*$_v(l, u)$ can simply be replaced by the distance value $d(v, u)$. The use of distance values in place of rank values can reasonably be expected to lead to better performances in practice; however, as has been previously noted, a distance-based formulation may not always be possible.

4.2.4 Cache Replacement Policy

In traditional setting, a cache stores and processes queries that are requested repeatedly. In a case of cache hit when result for a requested query is available in the cache: the cost of answering the query is very little. Contrarily, if the cache does not contain the result of a requested query, the result is fetched from the disk at a much higher cost. At this point the answer for this new query can be kept in the cache to answer future requests, but since the cache size is limited, answer for another query must be removed from the cache. A cache

replacement policy is a decision process that helps in retaining right objects in the cache to achieve the goal of answering as many requests as possible. Zipf's law and temporal locality principles help to make these decisions where least recently used, least frequently used and other replacement policies apply these principles to select the items for evictions. This process will automatically load the popular content in the cache after certain period of time. Selection of proper replacement method depends on the type of dataset and the traffic characteristics.

Traditional caching may not be effective in applications such as recommender systems where requests are unlikely to be repeated. For example, it is highly unlikely that a user asks for recommendations for an object again and again and hence, in that case almost none of the requests would hit an item already in a cache. In such case cache replacement policies could be very expensive and not much helpful in reducing the server load. However, a static cache, a cache without any replacement of the content, which is able to answer most of the queries could be more appropriate for these types of applications. Active caching approach in this dissertation also follows this approach where data is loaded in the cache only once and this data helps to answer cached as well as non-cached queries posed by the users and high cache hit rates can be achieved with out any replacement of the content.

4.3 Evaluation

As discussed earlier, existing active caching methods developed for boolean queries cannot in general be applied to handle recommender system queries, and thus a direct comparison between these methods and the proposed technique is not possible. For this reason, to evaluate the active caching approach for recommender system queries, this study instead compare their performance against those of traditional passive caching strategies in terms of four measures, the hit rate, the byte hit rate, the recall and the execution cost. Use of both hit rate and byte hit rate is due to the fact that the proposed solution requires more

space than the traditional caching approach due to the storage of inverted lists alongside standard result lists.

Partial order based active caching approach in general can be used with any nearest neighbor application. In order to test the performance of out proposed approach, different types of datasets have been selected . First, this study focuses on testing this approach with the two major categories of recommender systems i.e., content based and collaborative filtering. Second, it also focuses on other nearest neighbor applications e.g., image retrieval etc. Third, similarity functions used in nearest neighbor applications either result in a metric space e.g. Euclidean distance, or non-metric measures like cosine similarity. Datasets selected for evaluation in this study cover both content based and collaborative filtering recommendation techniques. Furthermore in these datasets similarity measure is also manipulated and both metric as well as non-metric similarity measures are used. Finally, to test nearest neighbor applications from various domains, datasets for digital libraries, images, network, intrusions, geographical data and rating data are used. Each of these dataset is detailed below.

4.3.1 Ranking Functions

Partial order approach discussed in this chapter uses ranking information to estimate answer for non-cached queries. Output is a top-k list of objects most reagent to the query object based on the information available in the cache. This relevancy or ranking is computed using the rank value of each object in all inverted list used to process the answer for an object. Aggregation of these rank values should be done carefully to get most accurate answer. This study uses four different ranking or aggregation methods to estimate top-k objects.

Min ranking function considers those objects most relevant whose rank value is most closer to the query object in any of the lists in which they co-occured with the query objects. Sum functions take into account all the occupancies of each object in these lists and sum

the rank values and k objects with highest sum values are returned. Avg function takes the average and returns those object with highest avg rank values. Skyline approach returns those objects which have higher rank in atleast one list and same in all other lists. In some cases it needs multiple iterations to return top-k skyline objects.

4.3.2 Experimental Results

In this study several tests were conducted using above mentioned datasets to check the performance of the proposed solution. Various proportions of the data were selected through a uniform selection at random and loaded to the cache. Every object v in the dataset was used in a query $Q(v, k)$ and posed against the cache for all the experiments.

Hit Rate: Partial order based approach test results showed substantially high hit rate when compared with a traditional caching approach. A traditional approach can only answer queries that are already in the cache, however, active caching can not only answer cached queries but also process answer for non-cached queries. For hit rate tests, various sizes of datasets were put in the cache and all possible queries, each object in the dataset was considered as a query, were executed. A hit is considered when answer can be provided from the cache.

For the ALOI dataset, Figure 4.3 shows the hit rate achieved by the active caching strategy for different choices of the standard list size λ. The proportion of items cached for this experiment varied between 10% and 100%. A traditional cache will always result in the same hit rate with varying list size because the list just kept to answer cached queries. However, active cache hit rate is directly proportional to the list size because it uses these lists to estimate answer for non-cached queries. In all cases, the hit rates were much higher with active caching than for passive caching, increasing very quickly with increasing list size. For the RCV1 dataset Figure 4.4, KDDCup dataset Figure 4.5, CoverType dataset

Figure 4.6 and Jester dataset Figure 4.7 also shows a very substantial improvement over passive caching.

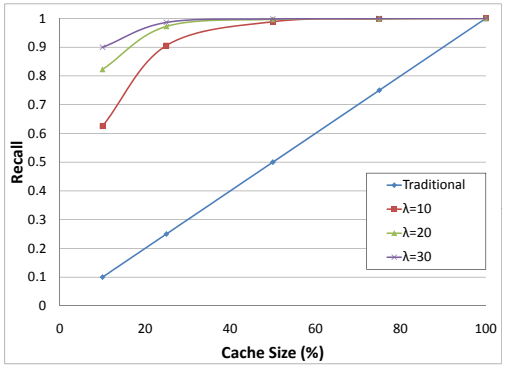

Figure 4.3 Hit rate for active caching across a range of cache sizes using dataset Aloi, with $\lambda = 10, 20, 30$.

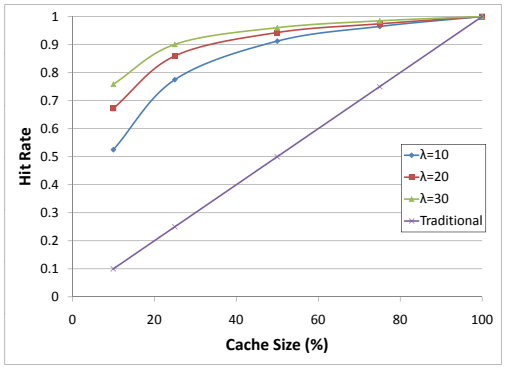

Figure 4.4 Hit rate for active caching across a range of cache sizes using dataset Reuters, with $\lambda = 10, 20, 30$.

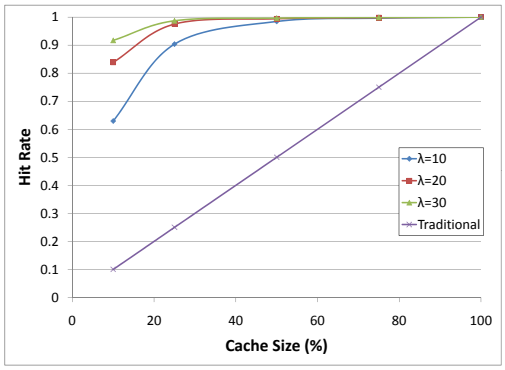

Figure 4.5 Hit rate for active caching across a range of cache sizes using dataset KDD Cup, with $\lambda = 10, 20, 30$.

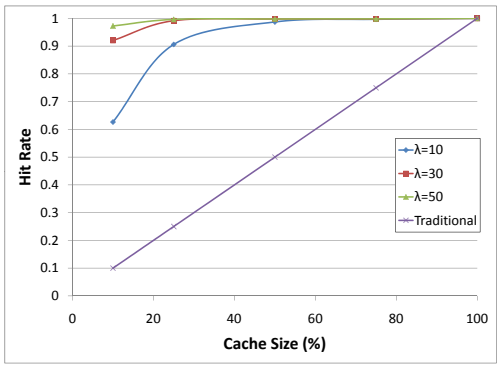

Figure 4.6 Hit rate for active caching across a range of cache sizes using dataset CoverType, with $\lambda = 10, 20, 30$.

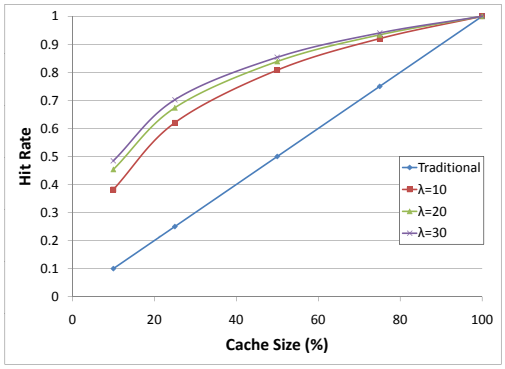

Figure 4.7 Hit rate for active caching across a range of cache sizes using dataset Jester, with $\lambda = 10, 20, 30$.

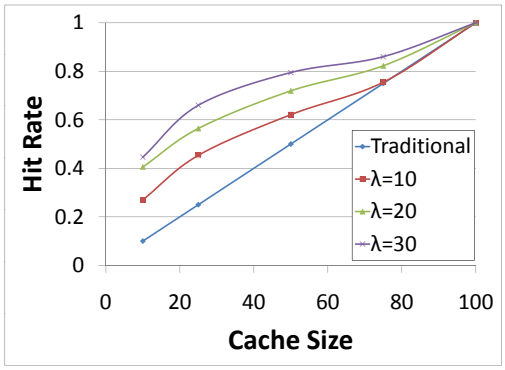

Figure 4.8 Hit rate for active caching across a range of cache sizes using dataset MovieLens, with $\lambda = 10, 20, 30$.

Experiments were conducted to see the hit rate in terms of total distinct objects available in the cache and number of queries that can be answered from the cache. Again active caching approach outperformed the traditional caching as showing in Figure 4.9

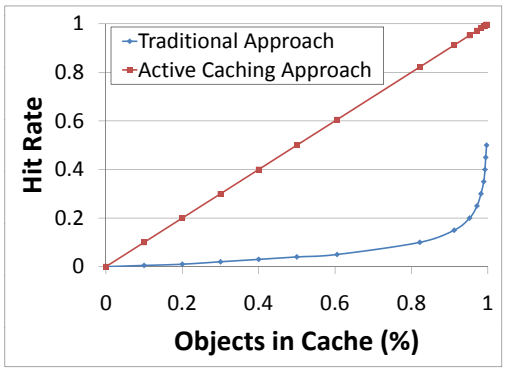

Figure 4.9 Hit rate for active caching across a range of distinct objects in cache using dataset ALOI, with $\lambda = 20$.

Content based data sets showed much higher hit rate than collaborative filtering cases. Possible reasons could be that the size of movielens dataset is very small, both users and movies, and also the available rating data is not equally distributed. The hit ratE mentioned above is based on the data size in the cache. Another advantage of the proposed active caching approach is that it utilizes cached list size effectively and shows high hit rate with increasing list size as demonstrated in Figure 4.3, Figure 4.4, Figure 4.5, Figure 4.6 and Figure 4.7. However, if the cache is considered in terms of cache size as oppose to data size, the result will be slightly different. Because the active cache uses twice the cache space of the traditional cache, caching 50% of the database will use as much space as caching 100% of the database in traditional cache. This is unlikely to happen in practice as the cache is designed for popular queries. In addition, database sizes are becoming too large (TB) to fit in a cache.

Byte Hit Rate: The proposed solution can require more space than the traditional caching approach due to the storage of inverted lists alongside standard result lists. For example, in a straightforward implementation in which integer variables are stored using 32 bits and floating point variables occupy 64 bits, each result item would be associated with 64 bits of storage: an integer object ID in the standard list, and an integer object ID in the inverted list. If traditional caching is performed with only object IDs being stored, only 32 bits would be required for each result list entry, and thus active caching would require approximately 2

times as much storage as the traditional implementation. However, if floating-point object distances were also required for the traditional implementation, the storage per entry would rise to 96 bits, leading to more storage cost as for active caching (since the active caching method would generate estimated similarity values with its own measure instead of relying on explicitly-stored distances). Nevertheless, it can be seen from Figures 4.10, 4.11, 4.12, 4.13 and 4.14 that for various cache sizes of data, the active caching solutions can answer substantially more queries than traditional caching even taking its potentially-increased memory requirements into account.

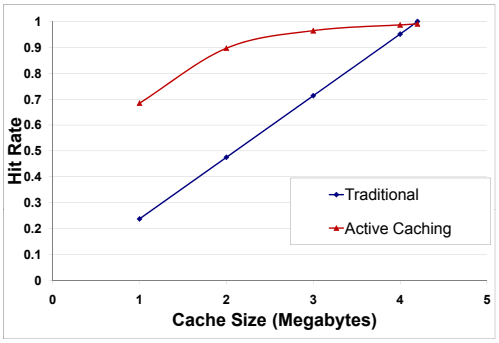

Figure 4.10 Byte Hit rate for active caching across a range of cache sizes in megabytes using dataset Aloi, with $\lambda = 10$.

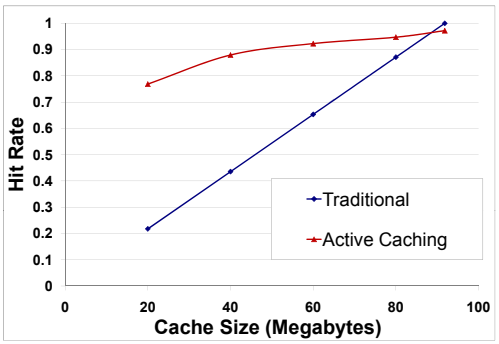

Figure 4.11 Byte Hit rate for active caching across a range of cache sizes in megabytes using dataset Reuters, with $\lambda = 30$.

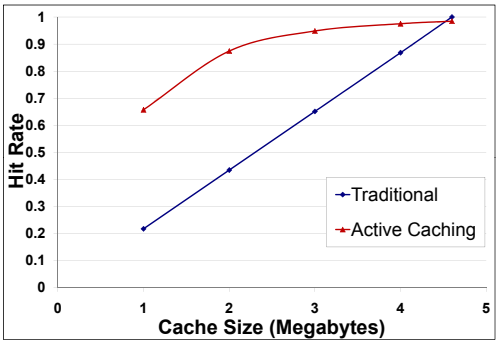

Figure 4.12 Byte Hit rate for active caching across a range of cache sizes in megabytes using dataset KDD Cup, with $\lambda = 10$.

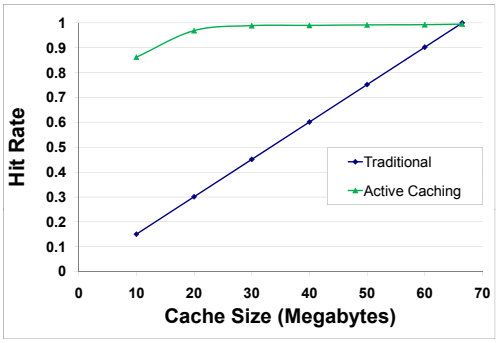

Figure 4.13 Byte Hit rate for active caching across a range of cache sizes in megabytes using dataset CoverType, with $\lambda = 30$.

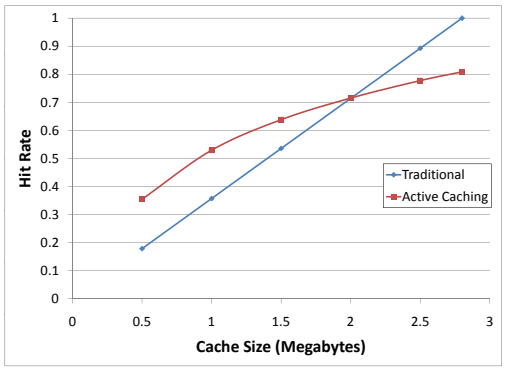

Figure 4.14 Byte Hit rate for active caching across a range of cache sizes in megabytes using dataset Jester, with $\lambda = 10$.

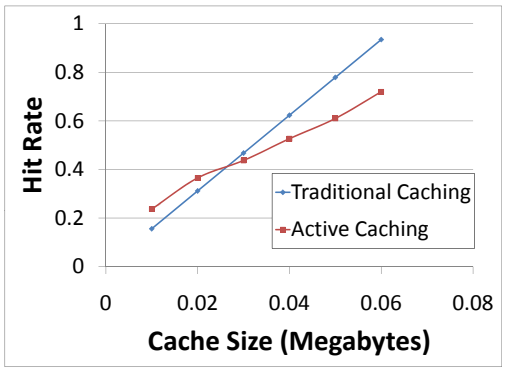

Figure 4.15 Byte Hit rate for active caching across a range of cache sizes in megabytes using dataset MovieLens, with $\lambda = 10$.

Recall: Recall and precision are the most commonly used accuracy measures. As mentioned earlier, in this case recall = precision, only recall test results are shown here using the above mentioned datasets. For each dataset, various sizes of data was loaded in the cache and all possible queries that can be answered from the cache were executed. The proposed active caching approach showed high recall values with various data sizes in the cache as shown in figures below.

Figures 4.16, 4.17, 4.18, 4.19 and 4.20 shows the cache recall values obtained over active caching hits for top-10 queries with standard list length $\lambda = 10$.

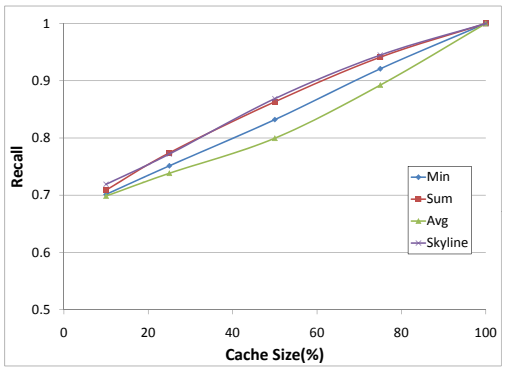

Figure 4.16 Average cache recall for active caching hits for the ALOI dataset, taken across a range of cache sizes with $k = \lambda = 10$.

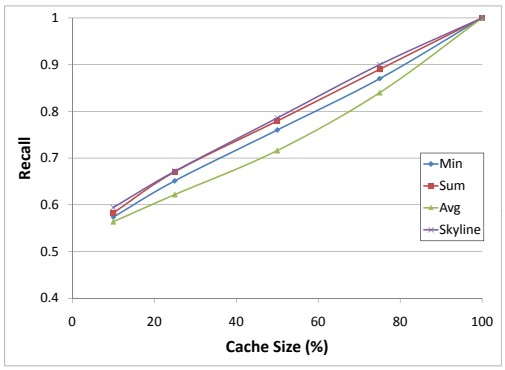

Figure 4.17 Average cache recall for active caching hits for the RCV1 dataset, taken across a range of cache sizes with $k = \lambda = 10$.

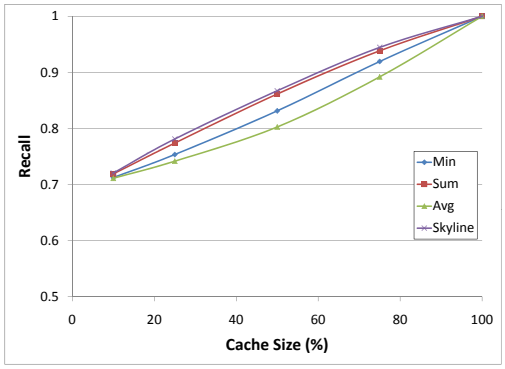

Figure 4.18 Average cache recall for active caching hits for the KDD dataset, taken across a range of cache sizes with $k = \lambda = 10$.

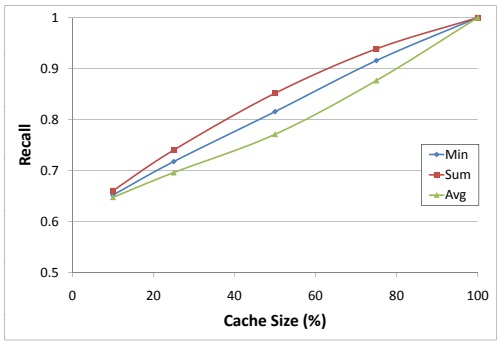

Figure 4.19 Average cache recall for active caching hits for the CoverType dataset, taken across a range of cache sizes with $k = \lambda = 10$.

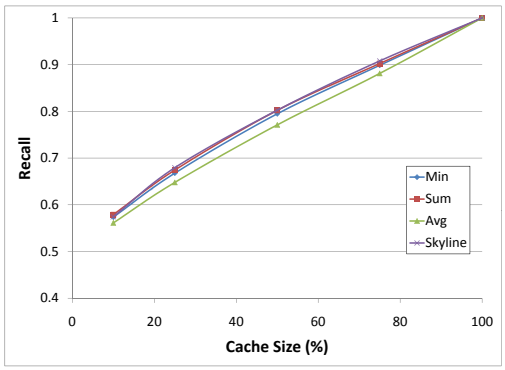

Figure 4.20 Average cache recall for active caching hits for the Jester dataset, taken across a range of cache sizes with $k = \lambda = 10$.

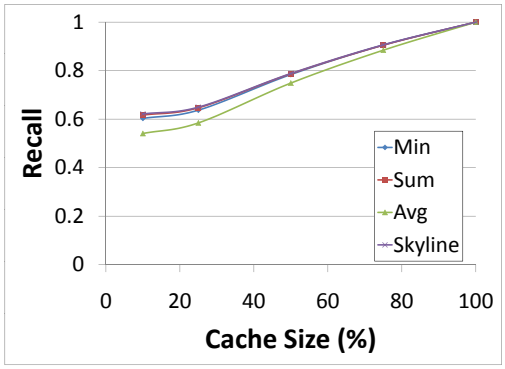

Figure 4.21 Average cache recall for active caching hits for the MovieLens dataset, taken across a range of cache sizes with $k = \lambda = 10$.

For all the datasets, the proposed solutions achieved very high recall values across the range of cache sizes. Sum and Skyline measures performed best in all the datasets with Skyline only slightly better than Sum measure. Together with hit rate and byte hit rate figures, these results show that for sufficiently-large cache sizes, very effective recall rates can be achieved while only rarely needing to access information on disk. For example, for the ALOI set with 25% of the data items cached, the proposed active caching variants answered approximately 98% of the queries, with an average recall of 0.75 and above. Even for the smallest cache size studied, 50% of the queries were answered with recall rates above 0.6, whereas the traditional caching approach would answer only 2.5% of the queries (albeit with recall 1.0).

To confirm the consistency of recall values for larger list sizes, list size k was also varied to see its impact on recall. Only test results for KDD and Aloi datasets are included

due to space limitations. These tests were conducted to test the recall by varying the size of the query result k together with the cached standard list size λ. Top-k queries were performed with $k = \lambda = 10$, 20, 30, 40 and 50 and cache size of 25%. The results, shown in Figure 4.22 and 4.23, did not vary significantly for different settings of λ, hence, confirming the reliability of the proposed approach with various size of λ.

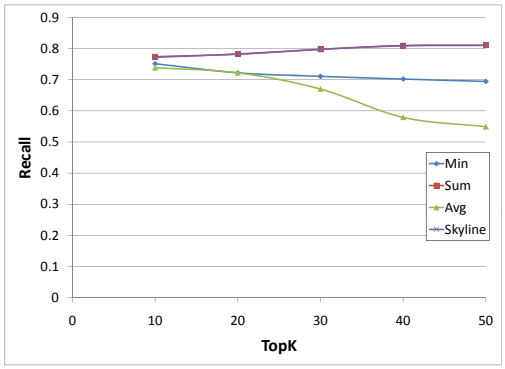

Figure 4.22 Average cache recall for top-k active caching hits using cache size of 25 for Aloi dataset, and across a range of list sizes $\lambda = k$.

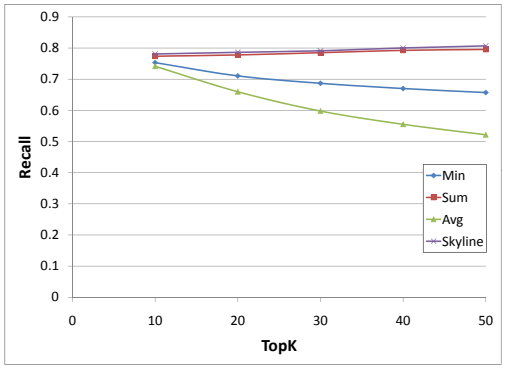

Figure 4.23 Average cache recall for top-k active caching hits using cache size of 25 for KDD dataset, and across a range of list sizes $\lambda = k$.

Above results again confirm the performance of Sum and Skyline measures as being best to use with the proposed approach. Both Avg and Min measure performance was degraded with the increase in list size however, recall rates for Sum and Skyline was very much consisted with the increase in list size.

For the remainder of the experiments in this section, this study concentrates on the estimation power of the proposed active caching methods, by forcing an active estimation

in which the standard list stored for the item at which the query is based is always ignored. In these experiments recall is calculated only for the estimated non-cached queries and cached queries are ignored.

To get further information on inverted list usage statistics, several experiments were conducted to show the relationship between observed average estimation recall rates and the sizes of the inverted lists associated with query items. Also information about number of objects was collected against each size of inverted list to further explore this relationship. In Figure 4.24 histogram shows the numbers of query items with inverted lists of a given size for Aloi dataset where as in Figure 4.25 the average recall values are plotted as a function of query inverted list size. For this experiment 25% objects were randomly selected and each having a standard list of size $\lambda = 20$.

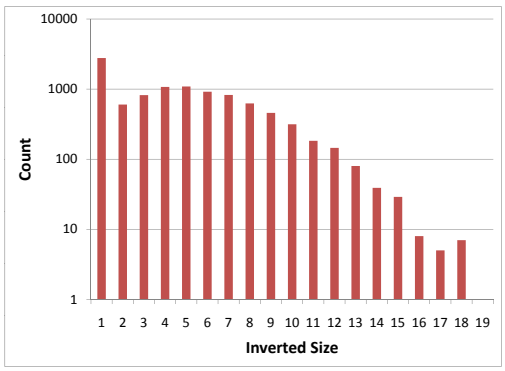

Figure 4.24 The histogram for ALOI shows the numbers of query items with inverted lists of a given size.

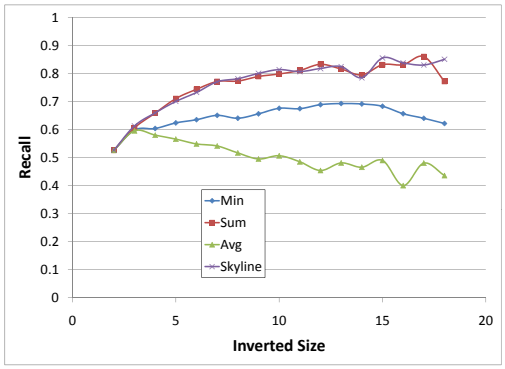

Figure 4.25 Average estimation recall values for active cache hits as a function of query inverted list size, for the ALOI dataset with cache size 25% and $k = \lambda = 20$.

The experimental results show consistently-high average estimation recall rates over all but the smallest inverted list sizes for both Sum and Skyline measures. Figure 4.25 suggest that better performances are achieved for longer query inverted list sizes. The high variation towards the end in recall for large inverted lists is due to the very small number of instances of these lists. A similar recall pattern can be seen for KDD Cup dataset in Figures 4.26 and 4.27

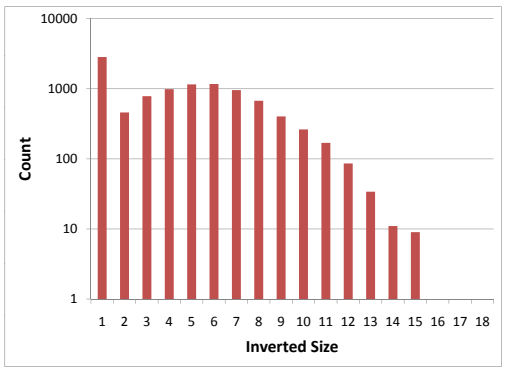

Figure 4.26 The histogram for KDD dataset shows the numbers of query items with inverted lists of a given size.

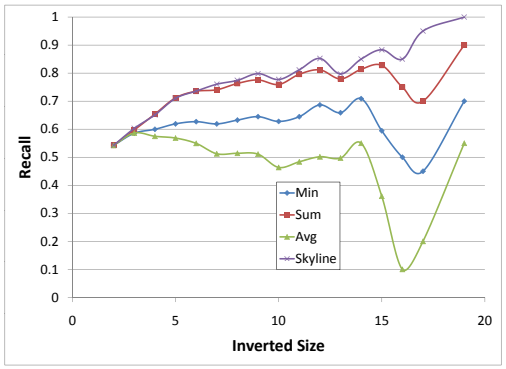

Figure 4.27 Average estimation recall values for active cache hits as a function of query inverted list size, for the KDD dataset with cache size 25% and $k = \lambda = 20$.

The above experiments on inverted lists show higher recall values for objects with larger inverted list sizes. Hence, overall recall for the cache can be improved by applying thresholds on inverted list size using the best performing measures i.e., Sum or Skyline. By using a threshold on inverted list size, active caching only generates a result only when the object satisfies a specified minimum value as evident from the Figures 4.25 and 4.27.

Another experiment was conducted to determine the influence of the choice of $Lambda$ on the recall rate of top-k active cache queries when the length of the cache standard lists is varied. In this test, the list lengths of $10, 20, 30, 40, 50$ were used to estimate top-30 results. and the cache size was again chosen to be 25%. Figure 4.28 and 4.29 shows a peak in performance for all methods with k between 30 and 40, with the Sum and Skyline methods again offering the greatest stability across the range.

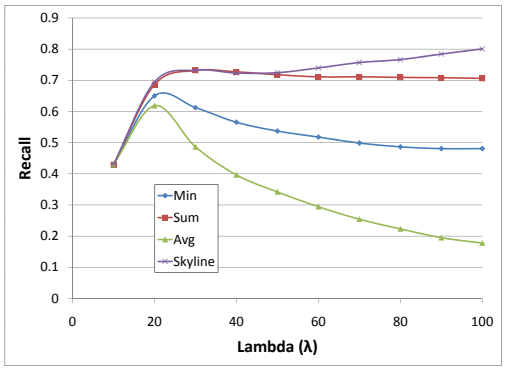

Figure 4.28 Average estimation recall rates of top-30 active cache hits for the ALOI dataset, plotted for a cache size of 25% against different standard list sizes.

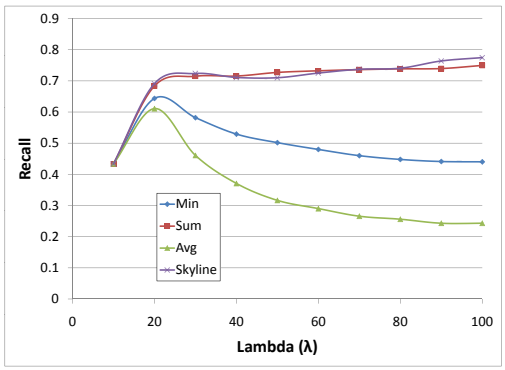

Figure 4.29 Average estimation recall rates of top-30 active cache hits for the KDD dataset, plotted for a cache size of 25% against different standard list sizes.

Execution Cost: In the experimental evaluation, efficiency of the proposed solution was tested in terms of the execution time of the queries posed. This test was conducted by

posing all possible similarity queries, and computing the execution time over each. For active caching, most of the queries will be answered from the cache and remaining will be answered from the database. For a traditional caching solution, fewer queries will be answered from the cache, and the remainder will be answered from the database. For comparison purposes, the cases where no cache is used were also considered, and an adjustment in which the size of the traditional cache is doubled. To test the cost of computation, cache was loaded with 25% of the data from the Aloi dataset and all the possible (110,250) queries were executed comparing the execution time of no caching , traditional caching and active caching . Active caching approach outperforms traditional cache [compared at 10,000 intervals]. Figure 4.30 shows the superior performance of the proposed approach in terms of CPU costs when compared with no caching and traditional caching approaches.

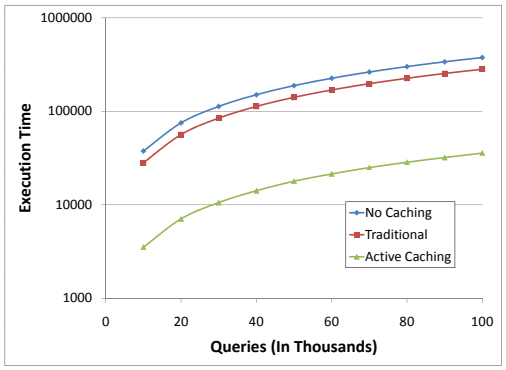

Figure 4.30 Query execution times for active caching across a range of query sizes, for the ALOI dataset with $k = \lambda = 30$. The execution costs (in milliseconds) for traditional caching, triple-sized traditional caching, and no caching is also shown.

The query execution time for the proposed solution as shown in the Figure 4.30 can be further improved. The execution time includes the sorting step before returning the results and it uses a simple bubble sort algorithm. An implementation of any better sorting algorithm can improve this time for the active caching approach; however, the execution time for traditional cache will remain same.

As mentioned above, if cache size is considered in terms of bytes of data, active cache uses twice the amount of size as in traditional cache. Figure 4.31 shows that even if traditional cache is loaded with twice the amount queries as in active cache, active cache out performs traditional caching approach in terms of CPU cost. A traditional cache was loaded with 50% data and active cache with 25% of the data from the same dataset (Aloi) and then executing all the possible queries (110250).

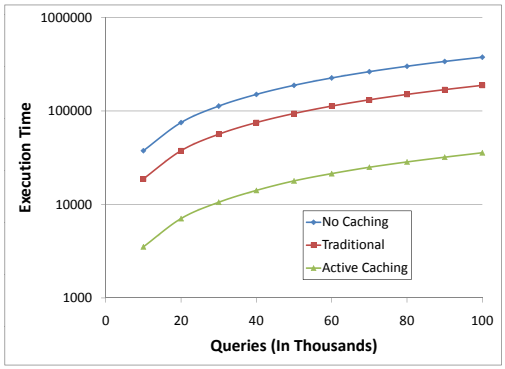

Figure 4.31 Performance test in terms of cpu cost, active cache outperforms traditional cache even when traditional cache is loaded with twice the amount queries as active cache.

Robustness: Robustness of the proposed solution is tested by introducing noise entries into the cached standard lists. In each test, cache lists were replaced by noise lists of the same length, generated by selecting objects from the full dataset uniformly at random. The proportion of cache lists replaced by noise lists was varied between 0% and 100%. The experimental results in Figure 4.32 and 4.33 shows a very strong linear relationship between the performance and the proportion of noise. The results indicate that relatively large amounts of noise can be tolerated while still providing very high recall rates.

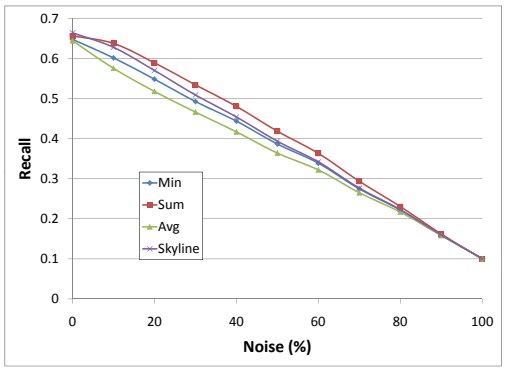

Figure 4.32 Average estimation recall rates for ALOI active cache hits, with a cache size of 25% and with $k = \lambda = 10$, plotted against the proportion of noise lists.

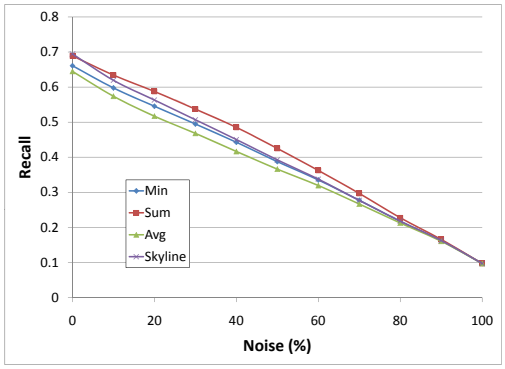

Figure 4.33 Average estimation recall rates for KDD active cache hits, with a cache size of 25% and with $k = \lambda = 10$, plotted against the proportion of noise lists.

4.4 Summary

Partial order based active caching approach mentioned above showed very strong overall performance and provides an intriguing answer to the question of cache management for recommender systems. The experimental results in the previous section show substantial improvement in the cache hit rate and the latency reduction in terms of execution costs as compared to traditional caching solutions. The proposed approach can not only answer queries that exactly match the queries in the cache but also computes answers for non-cached queries hence, the cache acts in a limited query processor and answers significantly higher number of queries than traditional caches.

Similar to any other caching solution, this approach also incurs an overhead in case of cache miss. This overhead is due to the fact that the first answer is checked in the cache and if it is not available then requested from the database. However, in active caching approach, the overhead is much lower because the number of cache misses are very low. The active caching solution has a marginal memory overhead due to the reverse lists but even if cache is considered in terms of byte size, it still provides significant performance gain.

This approach in general is suitable for both metric as well as non-metric distance measures and does not assume the queries lie in a metric space. However, it utilizes the monotonicity amongst the partial order lists to correctly compute the answer for non-cached queries. One of the limitation of this approach is that it can only achieve higher recall rates with the datasets having high level of monotonicity amongst the partial order lists. However, in practical it is not always difficult to assess the level of monotonicity in a dataset and the lower recall rates for non-cached queries could diminish the benefits of this approach. A possible solution to this limitation is to implement an approach which does not make use of monotonicity and thus can result in a higher recall for non-cached queries while maintaining the cache hit rate and execution cost. In the next chapter, shared neighbor approach is proposed which does not utilize monotonicity and results in a higher recall for non-cached queries.

CHAPTER 5

SHARED NEIGHBOR SIMILARITY MEASURE FOR ACTIVE CACHING

5.1 Introduction

Chapter 4 proposed a partial order based active caching approach that utilizes the monotonicity amongst partial order lists to compute the answer for non-cached queries. One of the limitation of partial order approach is that it can only achieve higher recall with the datasets having high level of monotonicity amongst the partial order lists. However, practically it is very difficult to estimate and improve the monotonicity in any dataset. In this chapter, a more general 'active caching' technique for K-NN queries is proposed which does not make use of monotonicity and can achieve higher recall while maintaining the hit rate. The solution is based on concepts from the relevant set correlation (RSC) clustering model, which measures the similarity between two objects in terms of the number of other objects in the common intersection of their neighborhoods. The intersection size of neighborhood sets has been used as the basis of the merge criteria of several heuristics for data clustering. These heuristics, collectively referred to as 'shared-neighbor' methods, use neighborhood intersection sizes in the estimation of local data density. These methods have the advantage of being more adaptable to variations in data distribution than methods that rely solely on distance measures. Examples of agglomerative shared-neighbor clustering algorithms include Jarvis and Patrick's method [54], ROCK [112], DBSCAN [84] and SNN [69]. More recently, a non-agglomerative clustering method, GreedyRSC [48], was proposed that uses shared-neighbor information to directly assess the quality of cluster candidates, and to rank the members of clusters in order of relevance.

The proposed approach provides a very general solution and can work with any application that results in a ranked list. General in a sense that it does not require the actual distance scores to compute the result from the cache. A more specific solution can

be implemented by using the triangular inequality if the underneath distance function is metric. Similarly, if the cached ranked lists show the property of monotonicity, partial order approach can be used to process non-cached queries. Although triangular inequality based and partial order based solution can be used, however, it will be a limitation and solution cannot be used with all those applications which results in a ranked list.

The active caching technique proposed in this chapter make use both of cached lists of objects in the neighborhood of query objects, as well as inverted lists derived from these neighbor lists. The lists used in this approach are fundamentally different from those used in [116],[53],[77]. Their approaches are designed specifically for keyword-based queries, whereas the proposed approach is much more general: it can be applied to any system that supplies ranked lists as query results, and relies only on the inherent ordering within these lists and their inversions.

This chapter refers to the preliminaries explained in Chapter 3. In the next section, a brief description of the original RSC model for clustering as well as supporting terminology and notation is presented. Section 5.3 lists the most commonly-used measures of shared neighbor information, and use techniques developed under the RSC model to derive similarity measures for active caching that take into account potential variation in the sizes of cached neighbor lists and/or inverted lists. Implementation section shows how these similarity measures can be efficiently implemented. Experiments are provided in Section 5.5 for various datasets, that show how the proposed active caching formulations can be surprisingly effective in terms of both recall and hit rate, even for relatively small cache sizes. Finally, the discussion is concluded in Section 5.6.

5.2 Relevant Set Correlation

Set correlation can be regarded as a special case of the well-known Pearson correlation of variable pairs. Every object of some universal set Ω can be associated with a coordinate of a vector space whose dimension is equal to the size of S. A subset A of Ω can be

represented by a zero-one characteristic vector in this space, where a coordinate value of 1 indicates that the corresponding object is a member of A, and a value of 0 indicates that the object does not belong to A. Even if no additional information is available regarding the nature of A and B, the relationship between A and B (and their underlying concepts) can be quantified in terms of the correlation between corresponding coordinates of their characteristic vectors.

For sequences of variables $(x_1, \ldots, x_{|\Omega|})$ and $(y_1, \ldots, y_{|\Omega|})$ with means $\bar{x}$ and $\bar{y}$, respectively, the standard Pearson sample correlation is given by the following formula [96]:

$$\begin{aligned} r &= \frac{\sum_{i=1}^{|\Omega|}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{|\Omega|}(x_i - \bar{x})^2 \sum_{i=1}^{|\Omega|}(y_i - \bar{y})^2}} \\ &= \frac{\sum_{i=1}^{|\Omega|} x_i y_i - |\Omega| \cdot \bar{x}\bar{y}}{\sqrt{(\sum_{i=1}^{|\Omega|} x_i^2 - |\Omega| \cdot \bar{x}^2)(\sum_{i=1}^{|\Omega|} y_i^2 - |\Omega| \cdot \bar{y}^2)}}. \end{aligned}$$

Applying the formula to the characteristic vectors of sets $A, B \subseteq \Omega$, and noting that

$$\sum_{i=1}^{|\Omega|} x_i^2 = \sum_{i=1}^{|\Omega|} x_i = |\Omega| \cdot \bar{x}$$

whenever $x_i \in \{0, 1\}$, the following set correlation formula can be obtained [48]:

$$\mathrm{R}_\Omega(A, B) = \frac{|\Omega| \cdot \mathbf{CM}(A, B) - \sqrt{|A| \cdot |B|}}{\sqrt{(|\Omega| - |A|)(|\Omega| - |B|)}},$$

where

$$\mathbf{CM}(A, B) = \frac{|A \cap B|}{\sqrt{|A| \cdot |B|}}$$

is the popular *cosine* similarity measure between A and B [96]. Note that when the sizes of A and B are fixed, the set correlation value tends to the cosine measure as the universal set size $|\Omega|$ increases.

Although the set correlation resembles to some extent the Spearman rank correlation and Kendall tau rank correlation coefficients appearing in the statistical literature [61], the

latter two are fundamentally unsuited for application to relevant sets. Given two ranked lists of items drawn from a common domain, Spearman rank correlation assigns to each item an ordered pair of values equal to the rank of the item with respect to each ranked list. The Spearman rank correlation value is simply the Pearson correlation applied to the collection of variable pairs. Compared to the set correlation, however, the Spearman rank correlation ascribes great importance to the magnitude of the difference between the ranks of a given item with respect to each list. The Kendall tau coefficient, on the other hand, is formulated in terms of the sign of the difference between the ranks of the item with respect to the two lists. For situations such as caching in which the available ranked lists span only a local neighborhood (the relevant set) and not the entire database, it is often the case that an individual item appears in one list and not the other, precluding the calculation of both Spearman and Kendall tau coefficients.

5.3 The CES Model

With traditional caching strategies, in processing a top-k query object v for which results have not already been cached, the information is retrieved directly from disk. However, if the query result for v can be reliably estimated using only cached information and without performing expensive disk access operations, the computational savings may be considerable. This section proposes the *Cache-Estimated Significance* (CES) model for the estimation of top-k query results using cached information, where parameters such as k, the cached neighbor list size λ, and the cache size are all allowed to vary. The model includes shared-neighbor similarity measures that, given any object $w \in S$, assesses the statistical significance of the relationship between v and w using only the information available in the cache. Using one of these measures, an approximation to the top-k query result for v can be generated by determining the k objects of S most closely related to v.

5.3.1 Shared-Neighbor Similarity Measures

Before presenting the details of the CES model and associated similarity measures, a spatially-motivated argument is given as to why shared neighbor information has the potential of indicating similarity relationships among data objects, even when the underlying distance values are not available.

Consider the situation in which the objects of a dataset S are embedded in a metric space with distance function *dist*, from which the relevancy ranking function Q is derived. Let x be any point in the space (not necessarily coincident with an object of S). With respect to any object $v \in S$, a *rank*(v, x) is defined to be the rank that a new object would be assigned if it were inserted into S at location x, with any tied distances broken in favor of x. Point x would also determine a standard relevant set $\mathrm{Q}(x, \lambda)$ and inverted relevant set $\mathrm{Q}^{-1}(x, \lambda)$, with respect to S using *dist*.

If x were allowed to migrate towards the location of v, the memberships of $\mathrm{Q}(x, \lambda)$ and $\mathrm{Q}^{-1}(x, \lambda)$ would tend progressively toward those of $\mathrm{Q}(v, \lambda)$ and $\mathrm{Q}^{-1}(v, \lambda)$, respectively, until they coincided at $x = v$. The relationships between $\mathrm{Q}(x, \lambda)$ and $\mathrm{Q}(v, \lambda)$ on the one hand, and $\mathrm{Q}^{-1}(x, \lambda)$ and $\mathrm{Q}^{-1}(v, \lambda)$ on the other, can serve as the foundation of a *rank measure* of the similarity between x and v. Each object of $\mathrm{Q}(x, \lambda) \cap \mathrm{Q}(v, \lambda)$ would support the contention that x and v were similar, and each object of $\mathrm{Q}(x, \lambda) \setminus \mathrm{Q}(v, \lambda)$ or $\mathrm{Q}(v, \lambda) \setminus \mathrm{Q}(x, \lambda)$ would work against it. The same would be true for inverted sets, with each of $\mathrm{Q}^{-1}(x, \lambda) \cap \mathrm{Q}^{-1}(v, \lambda)$ acting as a witness testifying to the similarity of x and v, and each object of $\mathrm{Q}^{-1}(x, \lambda) \setminus \mathrm{Q}^{-1}(v, \lambda)$ or $\mathrm{Q}^{-1}(v, \lambda) \setminus \mathrm{Q}^{-1}(x, \lambda)$ refuting it.

For the cache entry estimation problem, assume that the estimation of the top-k relevant set $\mathrm{Q}(v, k)$ for some object $v \notin C$. Even though only rank information is available and any spatial embedding and distance information is unknown, the spatial intuition described above can still be expected to apply in many (if not most) practical settings. The cached object set C would constitute a set of potential witnesses to the relationship between two objects v and w in S. However, as the cache generally contains only a small

fraction of the total possible standard relevant sets, any rank measure of similarity must rely primarily on the inverted relevant sets.

Just as different formulations of shared-neighbor criteria have been proposed for clustering applications, it is possible to devise many shared-neighbor similarity measures for estimation problem. Given an inverted cache $\mathcal{C}^{-1}(C, \lambda)$ and two inverted relevant sets $\mathrm{Q}_C^{-1}(v, \lambda)$ and $\mathrm{Q}_C^{-1}(w, \lambda)$, study focuses on the following three cache-estimated rank measures:

- the *intersection size* measure

$$\textit{SimInt}_{C,\lambda}(v, w) \triangleq |\mathrm{Q}_C^{-1}(v, \lambda) \cap \mathrm{Q}_C^{-1}(w, \lambda)|;$$

- the *set correlation* measure

$$\textit{SimCorr}_{C,\lambda}(v, w) \triangleq \mathrm{R}_C(\mathrm{Q}_C^{-1}(v, \lambda), \mathrm{Q}_C^{-1}(w, \lambda));$$

- and the *cosine* measure

$$\textit{SimCos}_{C,\lambda}(v, w) \triangleq \mathrm{CM}(\mathrm{Q}_C^{-1}(v, \lambda), \mathrm{Q}_C^{-1}(w, \lambda)).$$

The cosine measure is included here as a simplified alternative to the set correlation measure — in practical settings, the value of λ and the sizes of the inverted sets $\mathrm{Q}_C^{-1}(v, \lambda)$ and $\mathrm{Q}_C^{-1}(w, \lambda)$ are very much smaller than S and C, and the difference between $\textit{SimCos}_{C,\lambda}$ and $\textit{SimCorr}_{C,\lambda}$ is negligible.

For all three measures, a value of 1 indicates that all occurrences of v in the cached top-k query result lists coincide with occurrences of w (and vice versa), and thus that the cache strongly supports the association of v and w. On the other hand, values approaching 0 indicate little support for the association of v and w. The set correlation and cosine measures are not well-defined whenever $\mathrm{Q}_C^{-1}(v, \lambda) = \emptyset$ or $\mathrm{Q}_C^{-1}(w, \lambda) = \emptyset$. For these cases, the values of the measures are taken to be 0. For the set correlation measure, note that

$\mathrm{Q}_C^{-1}(v, \lambda)$ and $\mathrm{Q}_C^{-1}(w, \lambda)$ are subsets of C, and thus the characteristic vector descriptions take C to be the universal set, and not S.

5.3.2 Significance of Similarity Measures

The measures $\mathit{SimInt}_{C,\lambda}$, $\mathit{SimCos}_{C,\lambda}$ and $\mathit{SimCorr}_{C,\lambda}$ stated above may have different biases with respect to the cache size $|C|$, the relevant set size λ, and the sizes of the inverted relevant sets involved. The intersection size measure $\mathit{SimInt}_{C,\lambda}$ potentially favors those objects w having the largest inverted relevant sets $\mathrm{Q}_C^{-1}(w, \lambda)$, as no penalty is applied when members of this set do not appear in $\mathrm{Q}_C^{-1}(v, \lambda)$. The other two correlation measures compensate for this deficiency by normalizing the contributions with respect to the sizes of the inverted relevant sets. However, these two measures also admit the possibility of bias. In general, when making inferences involving Pearson correlation, a high correlation value alone is not considered sufficient to judge the significance of the relationship between two variables. When the number of variable pairs is small, it is much easier to achieve a high value by chance than when the number of pairs is large.

The RSC model for clustering was proposed as a way of correcting for the bias in shared-neighbor density measures for clustering applications [48]. The statistical significance of formulas involving set correlation values was tested against a 'hypothesis of randomness' — the assumption that each relevant set contributing to the density measure is independently generated via uniform random selection from among the available objects of S. In practice, of course, the relevant sets are far from random. However, this situation serves as a convenient reference point from which the significance of observed values of the measure can be assessed. Under the randomness hypothesis, the mean and standard deviation of the measure can be calculated, and standard scores (also known as Z-scores) [96] can then be generated and compared with one another. The more significant grouping would be the one whose standard score is highest — that is, the one whose correlation exceeds its expected value by the greatest number of standard deviations.

For the proposed cache entry estimation model, this study uses the 'hypothesis of randomness' to account for potential bias of $\textit{SimInt}_{C,\lambda}$ and $\textit{SimCorr}_{C,\lambda}$ due to the choice of standard cache list size λ, and the variations in the size of the inverted relevant sets that accompany the choice of λ. The analysis is similar to that of RSC significance in [48]. In that chapter, the expected value and variance of the set correlation was derived under the assumption that the sizes of the the sets were fixed, and that at least one set was chosen uniformly at random (without replacement) from among the objects of S. It requires that both sets be generated by random selection of objects, with each object selected independently with fixed probability. Let A be a set generated through independent random selection from the objects of set Ω, with each object present in A with probability $0 < p < 1$. Let B be a second set generated independently from Ω in the same manner as A. Then

- $\mathbf{E}[|A \cap B|] = p^2 \cdot |\Omega|$ and $\mathbf{Var}[|A \cap B|] = p^2(1-p^2) \cdot |\Omega|$,
- $\mathbf{E}[\mathrm{R}_\Omega(A,B)] = 0$ and $\mathbf{Var}[\mathrm{R}_\Omega(A,B)] = \frac{1}{|\Omega|-1}$.

Let $\underline{\textit{SimInt}}_{C,\lambda}(v,w)$ be a random variable representing the value of $\textit{SimInt}_{C,\lambda}(v,w)$ that would be achieved if all cached standard relevant sets were independently generated via the uniform random selection of λ objects from the dataset S. Similarly, let $\underline{\textit{SimCorr}}_{C,\lambda}(v,w)$ be a random variable representing the value of $\textit{SimCorr}_{C,\lambda}(v,w)$ that would be achieved under the same conditions. The probability p of an individual object $u \in C$ appearing in $\mathrm{Q}_C^{-1}(v,\lambda)$ would equal that of v appearing in $\mathrm{Q}(u,\lambda)$, which is $p = \frac{\lambda}{|S|}$. lemma 5.3.2 can then be applied with $\Omega = C$, $A = \mathrm{Q}_C^{-1}(v,\lambda)$, $B = \mathrm{Q}_C^{-1}(w,\lambda)$, and $p = \frac{\lambda}{|S|}$ to show that

- $\mathbf{E}[\underline{\textit{SimInt}}_{C,\lambda}(v,w)] = |C|\frac{\lambda^2}{|S|^2}$ and $\mathbf{Var}[\underline{\textit{SimInt}}_{C,\lambda}(v,w)] = |C|\frac{\lambda^2}{|S|^2}\left(1 - \frac{\lambda^2}{|S|^2}\right)$, and
- $\mathbf{E}[\underline{\textit{SimCorr}}_{C,\lambda}(v,w)] = 0$ and $\mathbf{Var}[\underline{\textit{SimCorr}}_{C,\lambda}(v,w)] = \frac{1}{|C|-1}$.

Given a set of cache objects C, the *cache-estimated correlation significance* of w relative to query v is defined as the standard score for $\textit{SimCorr}_{C,\lambda}(v,w)$ under the randomness

hypothesis:

$$\begin{aligned}
&\textit{ZCorr}_{C,\lambda}(v,w) \\
&\quad \triangleq \frac{\textit{SimCorr}_{C,\lambda}(v,w) - \mathbf{E}[\underline{\textit{SimCorr}}_{C,\lambda}(v,w)]}{\sqrt{\mathbf{Var}[\underline{\textit{SimCorr}}_{C,\lambda}(v,w)]}} \\
&\quad = \sqrt{|C|-1}\,\textit{SimCorr}_{C,\lambda}(v,w) \\
&\quad = \sqrt{|C|-1}\,\frac{|C|}{\sqrt{(|C|-|A|)(|C|-|B|)}} \\
&\qquad \cdot \left(\frac{|A\cap B|}{\sqrt{|A|\cdot|B|}} - \frac{\sqrt{|A|\cdot|B|}}{|C|}\right),
\end{aligned}$$

where $A = \mathrm{Q}_C^{-1}(v,\lambda)$ and $B = \mathrm{Q}_C^{-1}(w,\lambda)$. This indicates that as long as the cache size $|C|$ is kept constant, the correlation significance of w relative to v is equivalent to the correlation for the purposes of ranking.

The *cache-estimated intersection significance* of w relative to query v is defined as the standard score for $\textit{SimInt}_{C,\lambda}(v,w)$ under the randomness hypothesis:

$$\begin{aligned}
&\textit{ZInt}_{C,\lambda}(v,w) \\
&\quad \triangleq \frac{\textit{SimInt}_{C,\lambda}(v,w) - \mathbf{E}[\underline{\textit{SimInt}}_{C,\lambda}(v,w)]}{\sqrt{\mathbf{Var}[\underline{\textit{SimInt}}_{C,\lambda}(v,w)]}} \\
&\quad = \frac{\textit{SimInt}_{C,\lambda}(v,w) - np^2}{p\sqrt{n(1-p^2)}} \\
&\quad = \sqrt{|C|}\,\frac{|S|}{\sqrt{|S|^2-\lambda^2}} \\
&\qquad \cdot \left(\frac{|\mathrm{Q}_C^{-1}(v,\lambda)\cap \mathrm{Q}_C^{-1}(w,\lambda)|}{\frac{\lambda\cdot|C|}{|S|}} - \frac{\lambda}{|S|}\right).
\end{aligned}$$

The intersection significance somewhat resembles the correlation significance, in that it can be obtained from the latter as a result of the following substitutions (in order):

1. the factor $\sqrt{|C|}$ by $\sqrt{|C|-1}$;
2. the factor $\frac{|C|}{\sqrt{(|C|-|A|)(|C|-|B|)}}$ by $\frac{|C|}{\sqrt{|C|^2-|A|\cdot|B|}}$;

3. all occurrences of the set sizes $|A| = |\mathrm{Q}_C^{-1}(v, \lambda)|$ and $|B| = |\mathrm{Q}_C^{-1}(w, \lambda)|$ by their expected values, $\frac{\lambda \cdot |C|}{|S|}$.

5.3.3 Proof

Proof. Letting $\Omega = \{\omega_1, \omega_2, \ldots, \omega_n\}$, define zero-one random variables A_i and B_i for all $1 \leq i \leq n$, such that $A_i = 1$ if and only if $\omega_i \in A$, and $B_i = 1$ if and only if $\omega_i \in B$. Let $\bar{A} = \frac{1}{n}\sum_{i=1}^{n} A_i$ and $\bar{B} = \frac{1}{n}\sum_{i=1}^{n} B_i$ be the respective means of these random variables. Note that these random variables are all independent, and thus $\mathbf{E}[A_i A_j] = \mathbf{E}[A_i] \cdot \mathbf{E}[B_j]$ for all $1 \leq i < j \leq n$, and $\mathbf{E}[A_i B_j] = \mathbf{E}[A_i] \cdot \mathbf{E}[B_j]$ for all $1 \leq i, j \leq n$. Also, the fact that these variables are identically distributed implies that for any function $f : \{0, 1\} \to \mathbb{R}$, there is $\mathbf{E}[f(A_i)] = \mathbf{E}[f(A_j)] = \mathbf{E}[f(B_j)]$ for all $1 \leq i, j \leq n$.

The expected value of the size of the intersection of A and B is

$$\begin{aligned} \mathbf{E}[|A \cap B|] &= \mathbf{E}\left[\sum_{i=1}^{n} A_i B_i\right] \\ &= \sum_{i=1}^{n} \mathbf{E}[A_i] \cdot \mathbf{E}[B_i] \\ &= np^2. \end{aligned}$$

The variance of the size of the intersection of A and B is

$$\begin{aligned}
&\mathbf{Var}[|A \cap B|] \\
&= \mathbf{E}\left[(|A \cap B| - \mathbf{E}[|A \cap B|])^2\right] \\
&= \mathbf{E}\left[|A \cap B|^2 - 2np^2 \cdot \mathbf{E}[|A \cap B|] + n^2p^4\right] \\
&= \mathbf{E}\left[\left(\sum_{i=1}^{n} A_i B_i\right)^2\right] - n^2p^4 \\
&= \mathbf{E}\left[\sum_{i=1}^{n} A_i^2 B_i^2 + \sum_{i=1}^{n}\sum_{\substack{j=1 \\ j\neq i}}^{n} A_i B_i A_j B_j\right] - n^2p^4 \\
&= \sum_{i=1}^{n} \mathbf{E}[A_i] \cdot \mathbf{E}[B_i] \\
&\quad + \sum_{i=1}^{n}\sum_{\substack{j=1 \\ j\neq i}}^{n} \mathbf{E}[A_i] \cdot \mathbf{E}[B_i] \cdot \mathbf{E}[A_j] \cdot \mathbf{E}[B_j] - n^2p^4 \\
&= np^2 + n(n-1)p^4 - n^2p^4 = np^2(1-p^2).
\end{aligned}$$

The expected value of the correlation of A and B is

$$\begin{aligned}
&\mathbf{E}[\mathrm{R}_\Omega(A, B)] \\
&= \mathbf{E}\left[\frac{\sum_{i=1}^{n}(A_i - \bar{A})(B_i - \bar{B})}{\sqrt{\sum_{i=1}^{n}(A_i - \bar{A})^2 \cdot \sum_{i=1}^{n}(B_i - \bar{B})^2}}\right] \\
&= \sum_{i=1}^{n} \mathbf{E}\left[\frac{(A_i - \bar{A})(B_i - \bar{B})}{\sqrt{\sum_{i=1}^{n}(A_i - \bar{A})^2 \cdot \sum_{i=1}^{n}(B_i - \bar{B})^2}}\right] \\
&= \sum_{i=1}^{n} \mathbf{E}\left[\frac{(A_i - \bar{A})}{\sqrt{\sum_{i=1}^{n}(A_i - \bar{A})^2}}\right] \cdot \mathbf{E}\left[\frac{(B_i - \bar{B})}{\sqrt{\sum_{i=1}^{n}(B_i - \bar{B})^2}}\right] \\
&= \frac{1}{n}\mathbf{E}\left[\frac{\sum_{i=1}^{n}(A_i - \bar{A})}{\sqrt{\sum_{i=1}^{n}(A_i - \bar{A})^2}}\right] \cdot \mathbf{E}\left[\frac{\sum_{i=1}^{n}(B_i - \bar{B})}{\sqrt{\sum_{i=1}^{n}(B_i - \bar{B})^2}}\right] \\
&= \frac{1}{n} \cdot 0 \cdot 0 = 0.
\end{aligned}$$

The variance of the correlation of A and B is

$$
\begin{aligned}
&\mathbf{Var}[\mathrm{R}_\Omega(A,B)] \\
&= \mathbf{E}[(\mathrm{R}_\Omega(A,B) - \mathbf{E}[\mathrm{R}_\Omega(A,B)])^2] \\
&= \mathbf{E}[\mathrm{R}^2_\Omega(A,B)] \\
&= \mathbf{E}\left[\frac{\left(\sum_{i=1}^n (A_i-\bar{A})(B_i-\bar{B})\right)^2}{\sum_{i=1}^n (A_i-\bar{A})^2 \cdot \sum_{i=1}^n (B_i-\bar{B})^2}\right] \\
&= \mathbf{E}\left[\frac{\sum_{i=1}^n \sum_{j=1}^n (A_i-\bar{A})(A_j-\bar{A})(B_i-\bar{B})(B_j-\bar{B})}{\sum_{i=1}^n (A_i-\bar{A})^2 \cdot \sum_{i=1}^n (B_i-\bar{B})^2}\right] \\
&= \sum_{i=1}^n \sum_{j=1}^n \mathbf{E}\left[\frac{(A_i-\bar{A})(A_j-\bar{A})}{\sum_{h=1}^n (A_h-\bar{A})^2}\right] \cdot \mathbf{E}\left[\frac{(B_i-\bar{B})(B_j-\bar{B})}{\sum_{h=1}^n (B_h-\bar{B})^2}\right] \\
&= \sum_{i=1}^n \sum_{j=1}^n \mathbf{E}\left[\frac{(A_i-\bar{A})(A_j-\bar{A})}{\sum_{h=1}^n (A_h-\bar{A})^2}\right]^2,
\end{aligned}
$$

due to the independence and identical distributions of the random variables A_i and B_j for all $1 \leq i, j \leq n$.

Continuing,

$$
\begin{aligned}
&\mathbf{Var}[\mathrm{R}_\Omega(A,B)] \\
&= \sum_{i=1}^{n} \mathbf{E}\left[\frac{(A_i-\bar{A})^2}{\sum_{h=1}^{n}(A_h-\bar{A})^2}\right]^2 \\
&\quad + \sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n} \mathbf{E}\left[\frac{(A_i-\bar{A})(A_j-\bar{A})}{\sum_{h=1}^{n}(A_h-\bar{A})^2}\right]^2 \\
&= \frac{1}{n}\cdot\mathbf{E}\left[\frac{\sum_{i=1}^{n}(A_i-\bar{A})^2}{\sum_{h=1}^{n}(A_h-\bar{A})^2}\right]^2 \\
&\quad + \frac{1}{n(n-1)}\cdot\mathbf{E}\left[\frac{\sum_{i=1}^{n}\sum_{j=1_{j\neq i}}^{n}(A_i-\bar{A})(A_j-\bar{A})}{\sum_{h=1}^{n}(A_h-\bar{A})^2}\right]^2 \\
&= \frac{1}{n}+\frac{1}{n(n-1)} \\
&\quad \cdot\mathbf{E}\left[\frac{\sum_{i=1}^{n}(A_i-\bar{A})\left(\sum_{j=1}^{n}(A_j-\bar{A})-(A_i-\bar{A})\right)}{\sum_{h=1}^{n}(A_h-\bar{A})^2}\right]^2 \\
&= \frac{1}{n}+\frac{1}{n(n-1)}\cdot\mathbf{E}\left[\frac{(-1)\cdot\sum_{i=1}^{n}(A_i-\bar{A})^2}{\sum_{h=1}^{n}(A_h-\bar{A})^2}\right]^2 \\
&= \frac{1}{n}+\frac{1}{n(n-1)}\cdot(-1)^2 = \frac{1}{n-1}
\end{aligned}
$$

□

5.3.4 Ranking Functions

The use of standard scores as a measure of statistical significance can facilitate comparison across differing distributions. The cache-estimated significance measures, being standard scores, can thus be used to account for significance across such parameter choices as the dataset size $|S|$, the number of cache entries $|C|$, the cached list size λ, the query result size k, and the sizes of inverted sets. At query time, the parameters S and C can be considered to be fixed quantities, and the sizes of inverted sets vary according to the distribution of objects in the vicinity of the query object.

With regard to the number of cache entries, the correlation significance values $\mathit{ZCorr}_{C,\lambda}$ and $\mathit{ZInt}_{C,\lambda}$ both tend to increase as $|C|$ increases, due to the presence of the factor $\sqrt{|C|-1}$ in the former expression, and $\sqrt{|C|}$ in the latter. This accords well with the intuition that the greater the amount of cached information, the higher the quality of the query-result estimation under the model.

Under certain conditions, some of the secondary similarity measures proposed in this section turn out to be equivalent. In some practical settings, the cache and dataset sizes can greatly exceed the maximum sizes of the standard and inverted relevant sets; in these situations, the value of the set correlation measure $\mathit{SimCorr}_{C,\lambda}$ tends to that of the cosine measure $\mathit{SimCos}_{C,\lambda}$. When the cache size is considered to be fixed, $\mathit{ZCorr}_{C,\lambda}$ and $\mathit{SimCorr}_{C,\lambda}$ determine the same rankings of dataset objects, and can be used interchangeably. The latter measure is in fact more convenient to use as its values are restricted to the range $[-1,1]$. If the relevant set size λ is also taken to be fixed, the measure $\mathit{ZInt}_{C,\lambda}$ determines the same ranking of data objects as $\mathit{SimInt}_{C,\lambda}$. Similarly, the rankings due to $\mathit{ZInt}_{C,\lambda}(v,w)$ would be the same as those determined by the ratio between the inverted set intersection size and the average individual inverted set size:

$$\mathit{SimRatio}_{C,\lambda}(v,w) \triangleq \frac{|\mathrm{Q}_C^{-1}(v,\lambda) \cap \mathrm{Q}_C^{-1}(w,\lambda)|}{\lambda \cdot |C|/|S|}.$$

However, in general, neither of these equivalances hold if λ is allowed to vary. It should be noted that unlike $\mathit{SimInt}_{C,\lambda}$, $\mathit{SimCos}_{C,\lambda}$ and $\mathit{SimCorr}_{C,\lambda}$, the measure $\mathit{SimRatio}_{C,\lambda}$ may attain values exceeding 1.

If the values of $\mathrm{R}_C^{-1}(v,w,j)$ are readily available for all $1 \leq j \leq \lambda$, each object w can be scored with respect to query v according to the query size j for which it is most significant. Assuming that the dataset size $|S|$ and cache size $|C|$ are both taken to be constant, for the experimental evaluation to follow the following six ranking functions for the objects of S with respect to query object v were used:

- (derived from the significance of the intersection size measure *SimInt*) the ratio measure $\textit{SimRatio}_{C,\lambda}(v, w)$ and the *max-ratio* measure

$$\textit{MaxRatio}_{C,\lambda}(v, w) \stackrel{\triangle}{=} \max_{1 \le j \le \lambda} \textit{SimRatio}_{C,j}(v, w);$$

- (derived from the significance of the set correlation measure *SimCorr*) the set correlation measure

  $\textit{SimCorr}_{C,\lambda}(v, w)$, and the *max-correlation* measure

$$\textit{MaxCorr}_{C,\lambda}(v, w) \stackrel{\triangle}{=} \max_{1 \le j \le \lambda} \textit{SimCorr}_{C,j}(v, w);$$

- (derived from the limit of the set correlation measure *SimCorr* as the database and cache sizes increase) the cosine measure $\textit{SimCos}_{C,\lambda}(v, w)$, and the *max-cosine* measure

$$\textit{MaxCos}_{C,\lambda}(v, w) \stackrel{\triangle}{=} \max_{1 \le j \le \lambda} \textit{SimCos}_{C,j}(v, w).$$

5.4 Implementation

When standard cache $\mathcal{C}(C, \lambda)$, its inverted cache $\mathcal{C}^{-1}(C, \lambda)$, and their subcaches are all available in main memory, the intersection sizes $\textit{SimInt}_{C,j}(v, w)$ can be efficiently calculated for all $1 \le j \le \lambda$. A practical assumption that will help to lower the computational cost is that w need only be evaluated if the intersection size $\textit{SimInt}_{C,j}(v, w)$ is positive — otherwise, there is no information supporting the contention that w should be included in the estimated query result for v. Moreover, even if the intersection size is positive, a negative value of the correlation measure *MaxCorr* (or *SimCorr*) would indicate that the intersection between the inverted neighborhoods is less than what would be expected if the members of the original standard neighborhoods had been selected at random. Therefore, for any ranking function f chosen from among the six listed in the previous section, it is assumed that a threshold value $\varphi > 0$ has been supplied, and that only those $w \in S$ for

Table 5.1 Actual and Estimated Rankings of the Neighbors of Object 11 from the Example of Figure 5.1, with $\lambda = 8$, $|C| = 5$, and $|S| = 20$.

Euclidean		*SimCorr*		*MaxCorr*			*SimCos*		*MaxCos*			*SimRatio*		*MaxRatio*		
w	*dist*	w	*score*	w	*score*	j	w	*score*	w	*score*	j	w	*score*	w	*score*	j
11	0.000	11	1.000	11	1.000	4	11	1.000	11	1.000	4	11	1.500	11	1.500	8
7	1.000	3	1.000	3	1.000	8	3	1.000	3	1.000	8	1	1.500	1	1.500	8
3	1.202	7	0.667	7	1.000	4	7	0.817	7	1.000	4	3	1.500	3	1.500	8
12	1.647	0	0.408	12	1.000	4	1	0.775	12	1.000	4	5	1.500	5	1.500	8
0	1.944	8	0.408	14	1.000	4	5	0.775	14	1.000	4	7	1.000	7	1.000	4
5	1.961	10	0.408	0	0.612	6	0	0.577	1	0.775	8	17	1.000	12	1.000	4
14	2.000	12	0.408	8	0.612	6	8	0.577	5	0.775	8	0	0.500	14	1.000	4
1	2.119	14	0.408	10	0.612	6	10	0.577	0	0.707	6	8	0.500	17	1.000	8
8	2.499	1*	undef	5	0.408	7	12	0.577	8	0.707	6	9	0.500	0	0.667	6
13	2.832	5*	undef	15	0.167	7	14	0.577	10	0.707	6	10	0.500	8	0.667	6
15	3.102	9*	-0.167	1*	-0.167	6	17	0.577	17	0.577	8	12	0.500	10	0.667	6
17	3.201	15*	-0.167	2*	-0.250	4	9	0.408	15	0.500	7	13	0.500	15	0.571	7
10	3.432	17*	-0.408	4*	-0.250	4	15	0.408	9	0.408	8	14	0.500	9	0.500	8
9	3.655	2*	-0.612	6*	-0.250	4	13	0.333	13	0.333	8	15	0.500	13	0.500	8
2	4.425	4*	-0.612	9*	-0.250	4	2*	0.000	2*	0.000	4	2*	0.000	2*	0.000	4
4	4.432	6*	-0.612	16*	-0.250	4	4*	0.000	4*	0.000	4	4*	0.000	4*	0.000	4
6	4.541	16*	-0.612	18*	-0.250	4	6*	0.000	6*	0.000	4	6*	0.000	6*	0.000	4
16	4.791	18*	-0.612	19*	-0.250	4	16*	0.000	16*	0.000	4	16*	0.000	16*	0.000	4
19	5.175	19*	-0.612	13*	-0.408	4	18*	0.000	18*	0.000	4	18*	0.000	18*	0.000	4
18	5.268	13*	-0.667	17*	-0.408	5	19*	0.000	19*	0.000	4	19*	0.000	19*	0.000	4

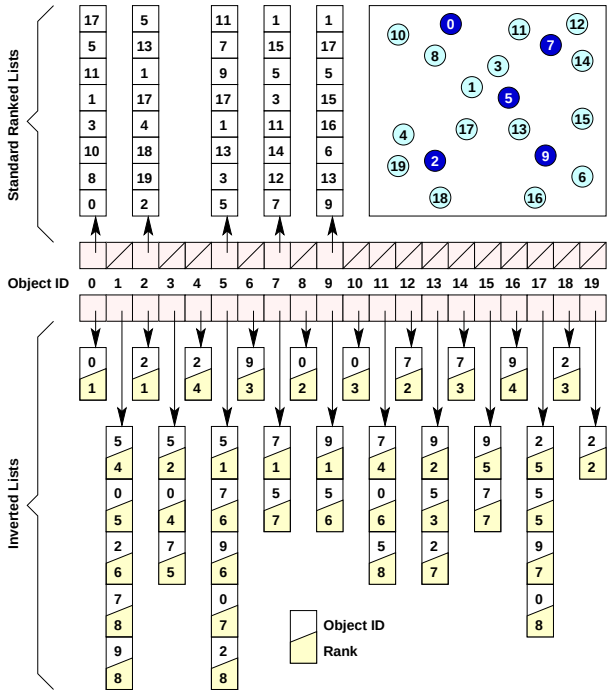

Figure 5.1 Cache data structures for a set of 20 objects in the 2-D plane. Top-8 lists are cached for 5 objects, with the Euclidean distance as the underlying ranking function.

which $f(v, w) \geq \varphi$ are eligible to appear in the estimated query result for v. If it turns out that fewer than the requested k objects are eligible, then the estimation is deemed to have failed, possibly necessitating an exact computation of the query result from information residing on disk.

The efficiency of the ranking process also depends on the storage of additional information with the inverted relevant sets:

1. For all $v \in C$, with each object $u \in \mathrm{Q}_C^{-1}(v, \lambda)$, the rank *rank*$(u, v)$ of v in $\mathrm{Q}(u, \lambda)$ is stored.

2. The objects of $\mathrm{Q}_C^{-1}(v, \lambda)$ are listed in non-decreasing order of these stored rank values.

With these preparations, the objects of set $\mathrm{Q}_C^{-1}(v, j)$ are simply those objects $u \in \mathrm{Q}_C^{-1}(v, \lambda)$ with stored rank value *rank*$(u, v) \leq j$, which can be read off from the head of the list in time proportional to the size of $\mathrm{Q}_C^{-1}(v, j)$.

In order to compute meaningful query result estimates, the cache size should be chosen so as to ensure that the inverted relevant sets are sufficiently large. This implies that $|C|$ and λ should be chosen so that the average inverted list size $\frac{\lambda \cdot |C|}{|S|}$ — the number of witnesses of the relationship between the query and its estimated result objects — exceeds some supplied threshold $\tau > 0$. Since $\lambda \cdot |C| \geq \tau \cdot |S|$, the main memory must be sufficently large to be able to store at least a constant amount of storage for every object in the dataset.

5.4.1 Cost

Computing the sizes of the intersections of inverted sets $\mathit{SimInt}_{C,j}(v, w)$ for every possible $w \in S$, if performed in the most straightforward manner, would be too costly an operation even with all necessary information resident in main memory. The computational cost can be reduced by first observing that for the majority of points $w \in S$, the intersection between $\mathrm{Q}_C^{-1}(v, j)$ and $\mathrm{Q}_C^{-1}(w, j)$ is empty. As it is reasonable to assume that only positive values of $\mathit{SimInt}_{C,j}$ are meaningful, it may limit the ranking effort to those w for which $\mathrm{Q}_C^{-1}(v, j) \cap \mathrm{Q}_C^{-1}(w, j) \neq \emptyset$. These objects are precisely those of the set $N(v, j) \stackrel{\triangle}{=} \bigcup_{u \in \mathrm{Q}_C^{-1}(v,j)} \mathrm{Q}(u, j)$, which can be constructed in time proportional to $O(j \cdot |\mathrm{Q}_C^{-1}(v, j)|)$, which is approximately $\tilde{O}(j^2)$. The overall time cost for ranking the objects could then be directly computed in time $O(j \cdot |\mathrm{Q}_C^{-1}(v, j)| + \sum_{w \in N(v,j)} |\mathrm{Q}_C^{-1}(w, j)|)$, or approximately $\tilde{O}(j^3)$. However, the computation time can be further reduced by computing the intersection sizes $|\mathrm{Q}_C^{-1}(v, j) \cap \mathrm{Q}_C^{-1}(w, j)|$ incrementally during the visitation of the members of $N(v, j)$, by noting that

$$|\mathrm{Q}_C^{-1}(v, j) \cap \mathrm{Q}_C^{-1}(w, j)| = |\{u \in \mathrm{Q}_C^{-1}(v, j) : w \in \mathrm{Q}(u, j)\}|.$$

The overall cost of ranking all objects can thus be reduced to $O(j \cdot |\mathrm{Q}_C^{-1}(v, j)| + |N(v, j)|)$, or approximately $\tilde{O}(j^2)$.

The values of the final ranking functions $\mathit{MaxRatio}_{C,\lambda}$, $\mathit{MaxCos}_{C,\lambda}$, and $\mathit{MaxCorr}_{C,\lambda}$ can be calculated by sequentially computing the respective values of $\mathit{SimRatio}_{C,j}$, $\mathit{SimCos}_{C,j}$,

and $\mathit{SimCorr}_{C,j}$ over the range $1 \leq j \leq \lambda$, and reporting the maximum observed. However, this would lead to a total time complexity of $O(\lambda^2 \cdot |\mathrm{Q}_C^{-1}(v,\lambda)| + \lambda \cdot |N(v,\lambda)|)$, or approximately $\tilde{O}(\lambda^3)$. Again, the computation time can be significantly reduced. Noting that

$$\begin{aligned}
&\left|\mathrm{Q}_C^{-1}(v,j+1) \cap \mathrm{Q}_C^{-1}(w,j+1)\right| \\
&= \left|\mathrm{Q}_C^{-1}(v,j) \cap \mathrm{Q}_C^{-1}(w,j)\right| \\
&\quad + \left|\left(\mathrm{Q}_C^{-1}(v,j+1) \setminus \mathrm{Q}_C^{-1}(v,j)\right) \cap \mathrm{Q}_C^{-1}(w,j+1)\right| \\
&\quad + \left|\mathrm{Q}_C^{-1}(v,j) \cap \left(\mathrm{Q}_C^{-1}(w,j+1) \setminus \mathrm{Q}_C^{-1}(w,j)\right)\right|,
\end{aligned}$$

the value of $|\mathrm{Q}_C^{-1}(v,j+1) \cap \mathrm{Q}_C^{-1}(w,j+1)|$ can be calculated using the precomputed value of $|\mathrm{Q}_C^{-1}(v,j) \cap \mathrm{Q}_C^{-1}(w,j)|$. Only the values of $|\left(\mathrm{Q}_C^{-1}(v,j+1) \setminus \mathrm{Q}_C^{-1}(v,j)\right) \cap \mathrm{Q}_C^{-1}(w,j+1)|$ and $|\mathrm{Q}_C^{-1}(v,j) \cap \left(\mathrm{Q}_C^{-1}(w,j+1) \setminus \mathrm{Q}_C^{-1}(w,j)\right)|$ need be explicitly computed in the transition from j to $j+1$. The total time, therefore, for computing $|\mathrm{Q}_C^{-1}(v,j) \cap \mathrm{Q}_C^{-1}(w,j)|$ for all $1 \leq j \leq \lambda$ is simply that of computing $|\mathrm{Q}_C^{-1}(v,\lambda) \cap \mathrm{Q}_C^{-1}(w,\lambda)|$, which as mentioned earlier is $O(\lambda \cdot |\mathrm{Q}_C^{-1}(v,\lambda)| + |N(v,\lambda)|)$, or approximately $\tilde{O}(\lambda^2)$.

5.4.2 Algorithm

The proposed method for the cache-estimated ranking of all objects of S with respect to a query object v is summarized below. In the pseudocode description, all objects are represented by IDs in the range $[0,\ldots,|S|)$. For any object $u \in C$ is denoted by $q(u,j)$ the ID of the object of rank j in the ranked list $\mathrm{Q}(u,\lambda)$. Also, $q^{-1}(v,j) = \{u \in C |\, v \in \mathrm{Q}(u,\lambda) \wedge \mathit{rank}(u,v) = j\}$ the set of IDs of objects of $\mathrm{Q}_C^{-1}(v,\lambda)$ having rank j. Note that the objects IDs of $\mathrm{Q}_C^{-1}(v,\lambda)$ are assumed to be sorted in terms of these ranks, and thus the objects of $q^{-1}(v,j+1)$ appear immediately after those of $q^{-1}(v,j)$ in $\mathrm{Q}_C^{-1}(v,\lambda)$.

Query

Input: query object ID v;

Output: object-correlation ordered pair list *Result*.

0. Initialization:

   (a) Static integer array for storing the number of visits to objects, assumed set to *RevIntersectCount*$[s]$=0 for all object IDs $0 \leq s < |S|$.

   (b) Static integer array for storing an object visit flag, assumed set to *VisitFlag*$[s] = 0$ for all object IDs $0 \leq s < |S|$.

   (c) Static real-valued array for storing object-to-query similarities *Score*$[s]$ for all object IDs $0 \leq s < |S|$.

   (d) Initialize lists *Visited* $\leftarrow \emptyset$ and *NewlyVisited* $\leftarrow \emptyset$.

1. Set list $L \leftarrow \mathrm{Q}_C^{-1}(v, \lambda)$ to contain the inverted relevant set for v, in the form of object IDs, with the entries sorted in non-decreasing order of the ranks they occupy in their neighbors' λ-relevant sets.

2. For all $1 \leq j \leq \lambda$ do:

   (a) Extract objects $L_j \leftarrow q^{-1}(v, j)$ from the head of L.

   (b) For all objects $u \in \bigcup_{i=1}^{j-1} L_i$ do:

      i. Let $w \leftarrow q(u, j)$.

      ii. If *RevIntersectCount*$[w] = 0$, then w has been visited for the first time overall. Perform the list insertion *Visited* $\leftarrow$ *Visited* $\cup \{w\}$.

      iii. Increment *RevIntersectCount*$[w]$ $\leftarrow$ *RevIntersectCount*$[w] + 1$.

      iv. If *VisitFlag*$[w] = 0$, then w has been visited for the first time in iteration j. Perform the list insertion *NewlyVisited* $\leftarrow$ *NewlyVisited* $\cup \{w\}$, and set *VisitFlag*$[w] \leftarrow 1$.

(c) For all objects $u \in L_j$ do:

i. For all objects $w \in \mathrm{Q}(u, j)$:

A. If *RevIntersectCount*$[w] = 0$, then w has been visited for the first time overall. Perform the list insertion *Visited* $\leftarrow$ *Visited* $\cup \{w\}$.

B. Increment *RevIntersectCount*$[w]$

$\leftarrow$ *RevIntersectCount*$[w] + 1$.

C. If *VisitFlag*$[w] = 0$, then w has been visited for the first time in iteration j. Perform the list insertion *NewlyVisited* $\leftarrow$ *NewlyVisited* $\cup$ $\{w\}$, and set

VisitFlag$[w] \leftarrow 1$.

(d) For all objects $w \in$ *NewlyVisited* do:

i. Let *SimInt*$_{C,j}(v, w) \leftarrow$ *RevIntersectCount*$[w]$.

ii. Compute

$$\textit{Temp} \leftarrow \begin{cases} \textit{SimRatio}_{C,j}(v, w) \\ \textit{SimCos}_{C,j}(v, w) \\ \textit{SimCorr}_{C,j}(v, w) \end{cases}$$

from *SimInt*$_{C,j}(v, w)$, as appropriate.

iii. If *Temp* $>$ *Score*$[w]$ then update *Score*$[w] \leftarrow$ *Temp*.

iv. Reset *VisitFlag*$[w] \leftarrow 0$, and delete

NewlyVisited $\leftarrow$ *NewlyVisited* $\setminus \{w\}$.

3. For all objects $w \in$ *Visited* do:

(a) Append the object-correlation ordered pair

Result $\leftarrow$ *Result* $\cup \{(w,$ *Score*$[w])\}$.

(b) Reset *RevIntersectCount*$[w] \leftarrow 0$, and delete

Visited $\leftarrow$ *Visited* $\setminus \{w\}$.

4. Sort the ordered pairs of list *Result* in non-increasing order of the values, and return the list. Ties can be broken arbitrarily, with the exception that v is given priority over any other object $w \neq v$ in S.

5.5 Evaluation

5.5.1 Performance Measures and Datasets

As already discussed in Chapter 2, existing active caching methods developed for Boolean queries cannot in general be applied to handle similarity queries, and thus a direct comparison between these methods and the proposed techniques is not possible. For this reason, to evaluate the active caching strategies for top-k similarity queries proposed in Section 5.3, their performance is compared against those of passive caching strategies in terms of two measures, the *hit rate* and the *recall*. Recall performance of the shared-neighbor based approach and partial order based approach is also compared. Hit rate is not compared because both approaches answer same number of queries from the cache.

Overall estimation power of the proposed active caching methods is also assessed. To do this, the recall that would be achieved for the query result estimated by the active caching method is reported, without checking whether the true query result is explicitly stored in the cache. To distinguish the two interpretations of recall, this latter interpretation is referred as the *estimation recall*, and the former (usual) interpretation as the *cache recall*.

Six data sets were used for experimentation, each of these are described in Chapter 3.

All six methods — *SimCorr*, *SimCos*, *SimRatio*, *MaxCorr*, *MaxCos* and *MaxRatio* — were implemented in Microsoft C♯, and tested on an IBM desktop computer with an Intel Xeon 3.0 GHz processor, with 8 GB of main memory, and running the Windows Server 2003 operating system. For both data sets, the standard cache lists and inverted lists were managed using Microsoft SQL Server.

5.5.2 Experimental Results

For all six datasets, several top-k query-by-example retrieval experiments were conducted, over a variety of cache and list sizes, and of choices of k. For each experiment, subsets of the data objects were selected uniformly at random for inclusion in the standard cache, at different proportions of the total dataset size. The cached information consisted of the standard neighbor list for each cache item, and their associated inverted lists. After setting up the cache, each of the objects of the full dataset was used as the basis of a query-by-example operation. In all the experiments that follow, experimental results are shown only for the *SimCorr*, *MaxCorr*, *SimRatio* and *MaxRatio* methods, as the values *SimCos* and *MaxCos* were virtually identical to those of *SimCorr* and *MaxCorr* for even the smallest of the cache sizes considered. Finally, the experiments were conducted to compare the shared neighbor approach with partial order approach using only *SimRatio* measure which performed best in the earlier experiments.

Efficiency: For the ALOI dataset, Figure 5.2 shows the hit rate achieved by the active caching strategy for different choices of the standard list size λ. The proportion of items cached for this experiment varied between 2.5% and 100%. In all cases, the hit rates were much higher with active caching than for passive caching, increasing very quickly with increasing list size. For the RCV1 dataset, Figure 5.3 also shows a very substantial improvement over passive caching.

The proposed solution can require more space than the traditional caching approach due to the storage of inverted lists alongside standard result lists. For example, in a straightforward implementation in which integer variables are stored using 32 bits and floating point variables occupy 64 bits, each result item would be associated with 96 bits of storage: an integer object ID in the standard list, and an integer object ID plus an integer rank in the corresponding inverted list entry. If traditional caching is performed with only object IDs being stored, only 32 bits would be required for each result list

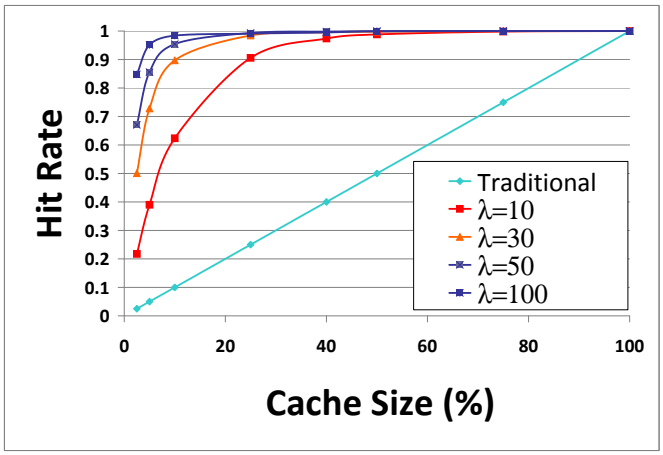

Figure 5.2 Hit rate for active caching across a range of cache sizes using the ALOI data set.

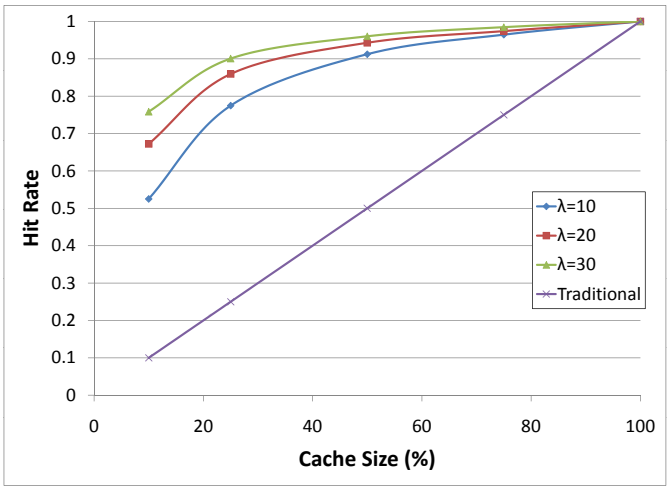

Figure 5.3 Hit rate for active caching across a range of cache sizes using dataset RCV1.

entry, and thus active caching would require approximately 3 times as much storage as the traditional implementation. However, if floating-point object distances were also required for the traditional implementation, the storage per entry would rise to 96 bits, leading to approximately the same storage cost as for active caching (since the active caching method would generate estimated similarity values with its own measure instead of relying on explicitly-stored distances). Nevertheless, it is can be seen from Figures 5.2 and 5.3 that for cache sizes of approximately less than one-third of the data, active caching solutions can answer substantially more queries than traditional caching even taking its potentially-increased memory requirements into account.

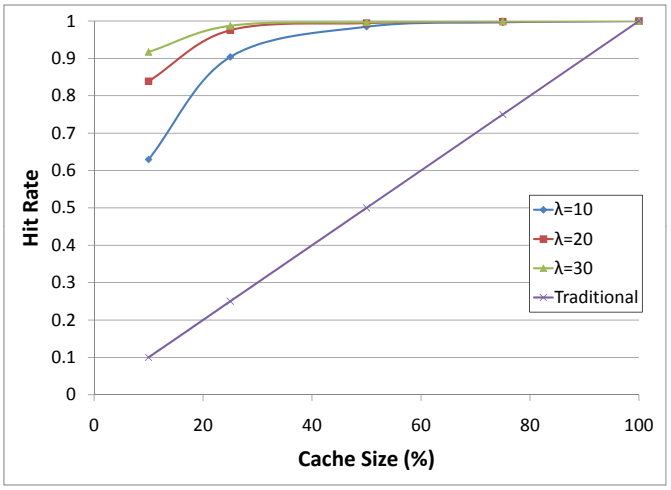

Figure 5.4 Hit rate for active caching across a range of cache sizes using dataset KDD.

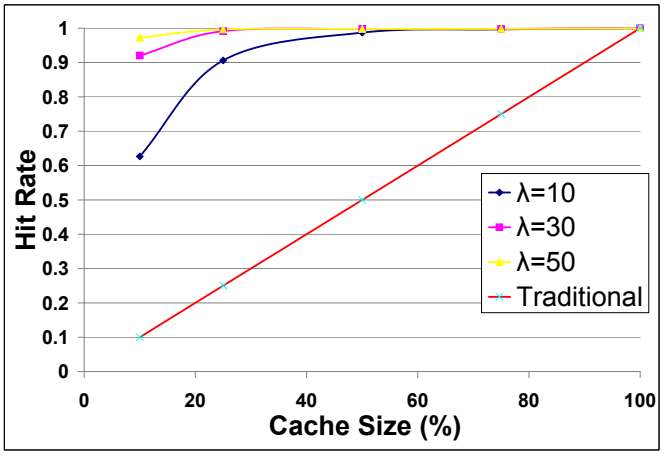

Figure 5.5 Hit rate for active caching across a range of cache sizes using CoverType dataset.

Effectiveness: Figures 5.8 and 5.9 shows the cache recall values obtained over active caching hits for top-30 queries with standard list length $\lambda = 30$. For both the ALOI and RCV1 datasets, The proposed solutions achieved very high recall values across the range of cache sizes. Together with Figures 5.2 and 5.3, these results show that for sufficiently-large cache sizes, very effective recall rates can be achieved while only rarely needing to access information on disk. For example, for the ALOI set with 25% of the data items cached, the proposed active caching variants answered approximately 98% of the queries, with an average recall of 0.8 and above. Even for the smallest cache size studied, 50% of the queries

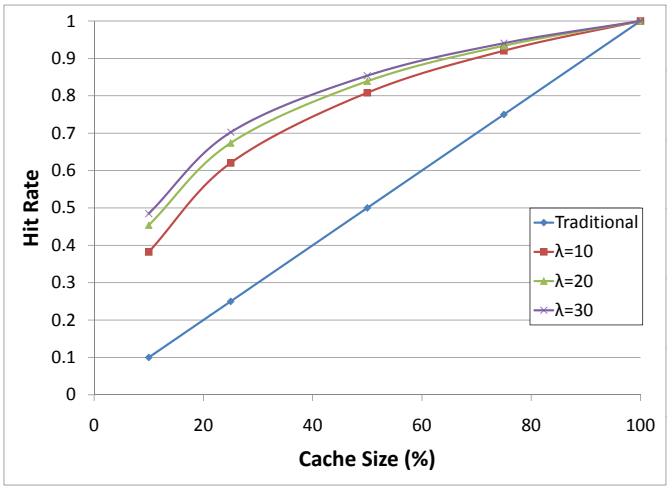

Figure 5.6 Hit rate for active caching across a range of cache sizes using CoverType dataset.

were answered with recall rates above 0.65, whereas the traditional caching approach would answer only 2.5% of the queries (albeit with recall 1.0). Recall rates for various datasets are shown in 5.8, 5.9, 5.10, 5.11, 5.12, 5.13.

Next, a similar experiment was performed on the ALOI dataset for one representative measure, *SimRatio*, this time varying the size of the query result k together with the cached standard list size λ. Top-k queries were performed with $k = \lambda = 10$, 30, 50 and 100. The results, shown in Figure 5.14, did not vary significantly for different settings of λ; however, the same general dependence on the cache size was observed as in the previous experiment.

For the remainder of the experiments in this section, this section concentrates on the estimation power of active caching methods, by forcing an active estimation in which the standard list stored for the item at which the query is based is always ignored.

Experiments were conducted to show the relationship between observed average estimation recall rates and the sizes of the inverted lists associated with query items. In Figure 5.15 the average recall values for the ALOI dataset are plotted as a function of query inverted list size, together with a histogram showing the numbers of query items involved. Top-30 queries were performed based at all 110,250 ALOI images. 25% of the ALOI images were selected at random for inclusion in the cache, each having a standard list

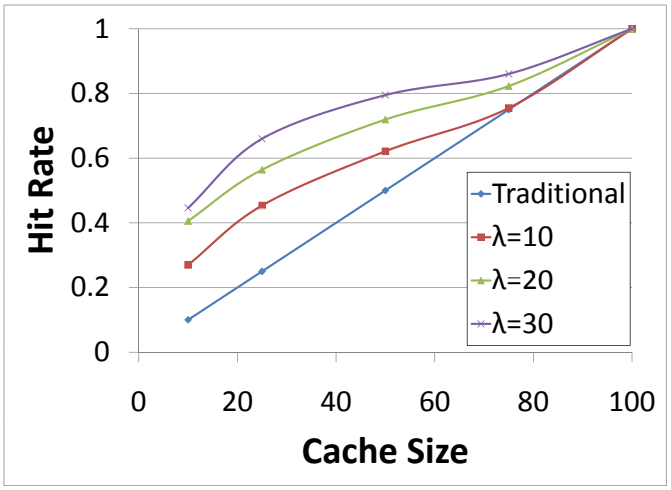

Figure 5.7 Hit rate for active caching across a range of cache sizes using CoverType dataset.

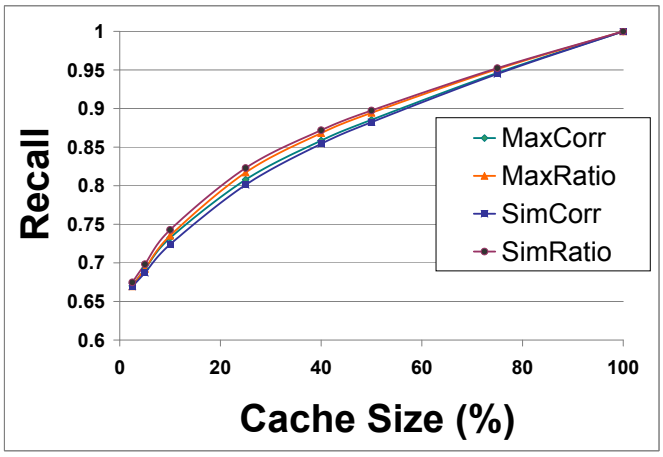

Figure 5.8 Average cache recall for active caching hits for the ALOI dataset, taken across a range of cache sizes with $k = \lambda = 30$.

of size $\lambda = 30$. The experimental results show consistently-high average estimation recall rates over all but the smallest inverted list sizes, with *MaxRatio* and *SimRatio* performing slightly better than *MaxCorr* and *SimCorr* over most of the range. They suggest that better performances are achieved for the largest query inverted list sizes.

The effect on estimation recall is also measured when minimum thresholds are applied to *SimRatio*, *MaxRatio*, *SimCorr* and *MaxCorr* scores. Again, top-30 queries were performed for the ALOI dataset, with 25% of the images cached and $\lambda = 30$. In this experiment, active caching was used to generate a result only when each item in the estimated top-k result list

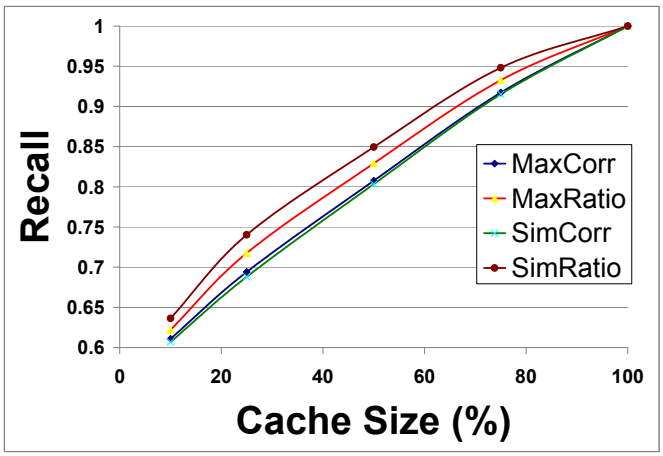

Figure 5.9 Average cache recall for active caching hits for the RCV1 dataset, taken across a range of cache sizes with $k = \lambda = 30$.

achieved a score higher than a specified minimum threshold value. Figure 5.16 shows plots of the average recall for the four active caching methods, against a range of score threshold values. The results show a positive influence between the minimum scores obtained and the recall rates achieved.

For each of the top-k query experiments presented above, the standard list size λ was set equal to k. In order to provide insights into the effect of λ and k on the performance of the proposed active caching methods, a further experiment was conducted in which λ was varied while fixing k. Once again, a cache size of 25% was chosen from the ALOI dataset, and k chosen to be 30; the standard list sizes were set at $\lambda = 10$, 20, 30, 50 and 100. The results of the experiment are displayed in Figure 5.17. All of the caching methods performed best for the case $\lambda = k = 30$. For smaller values of λ, the performance degenerated markedly. For larger values, the recall rates remained high, although the *SimRatio* and *SimCorr* performances showed substantially more degradation than those of *MaxRatio* and *MaxCorr*. Although *SimRatio* performed marginally better than *MaxRatio* for the case $\lambda = k = 30$, the latter method achieved the best overall performance when larger list sizes were used.

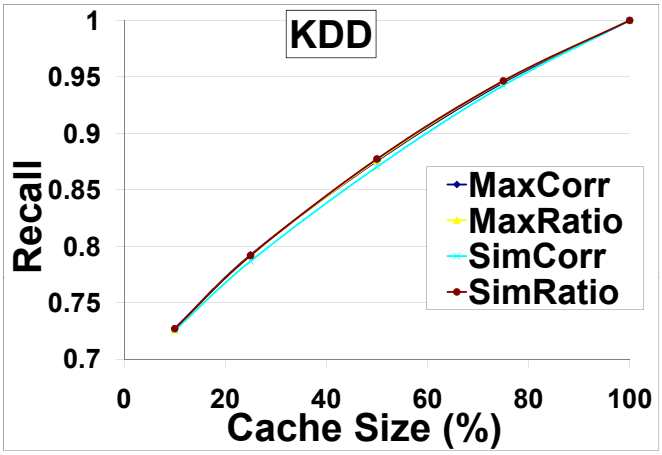

Figure 5.10 Average cache recall for active caching hits for the KDD dataset, taken across a range of cache sizes with $k = \lambda = 30$.

Comparison with Partial Order Approach: Partial order based approach presented in the last chapter uses monotonicity amongst partial order lists to compute answers for non-cached queries. Due to the high dependency on monotonicity, partial order based approach results in lower recall rates for datasets having lower levels of monotonicity. Shared neighbor approach presented in this chapter eliminates the need for monotonicity and just uses the witnesses to compute the answer for non-cached queries. Both of these approaches use similar architecture in terms of processing the number of queries from the cache, the hit rate and execution cost is exactly similar and not compared here. Main difference in terms of output is in the recall as both approaches use difference strategies to compute the answer for non-cached queries. Cached queries are always answered with a perfect recall of 1 in both approaches hence, the recall results are only shown for non-cached queries. Figure 5.18 shows a comparison of recall rates for partial order based approach and shared neighbor approach using ALOI dataset.

Similar experiment is conducted for Reuters, KDD, CoverType, Jester and MovieLens datasets. Similarly, Figure 5.19, Figure 5.20, Figure 5.21, Figure 5.22 and Figure 5.23 shows a comparison of recall rates for partial order based approach and shared neighbor approach for other datasets.

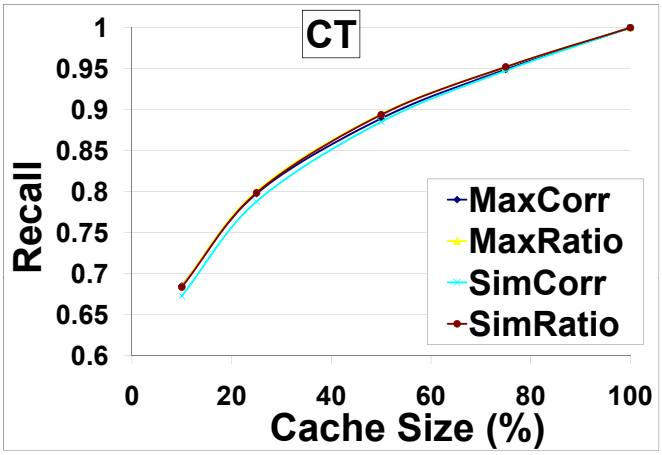

Figure 5.11 Average cache recall for active caching hits for the CoverType dataset, taken across a range of cache sizes with $k = \lambda = 30$.

KDD and CoverType datasets showed small improvement in recall using shared neighbor approach. To further confirm that this improvement is significant T-tests results for both of these datasets and the result showed that the improvement is significant. Since the P-value = 0 for all cases, then there is a very strong evidence against H0 (states that the no difference between the two approaches). Equivalently, the data strongly recommend that the shared neighbor approach provides a higher average recall than the partial order approach.

Robustness: Proposed solution is tested for robustness by introducing noise entries into the cached standard lists. In each test, ALOI cache lists were replaced by "noise lists" of the same length, generated by selecting objects from the full dataset uniformly at random. The proportion of cache lists replaced by noise lists was varied between 0% and 100%. The experimental results in Figure 5.24 shows a very strong linear relationship between the performance and the proportion of noise. The results indicate that relatively large amounts of noise can be tolerated while still providing very high recall rates.

It is possible that for some caching applications, the stored result lists may not all be of the same length, as has been assumed so far in the experimentation. Accordingly, an

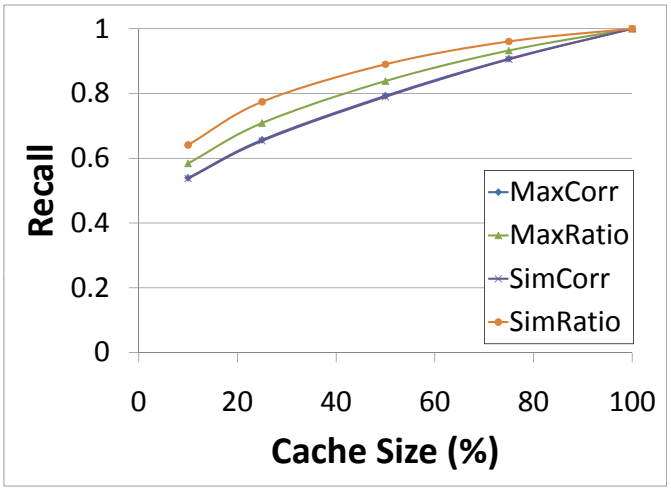

Figure 5.12 Average cache recall for active caching hits for the Jester dataset, taken across a range of cache sizes with $k = \lambda = 20$.

experiment is conducted on the ALOI dataset in which top-30 query result estimation was performed, with a varying proportion of the standard cache list lengths selected uniformly at random between 1 and 100, and the remainder of the lists having lengths fixed at 30. The cache size was chosen to be 25%. The results of the experiment are shown in Figure 5.25. Although some degradation of performance was observed for the *SimCorr* and *SimRatio* measures with increasing proportions of variable-sized lists, the estimation recall rates of *MaxCorr* and *MaxRatio* were remarkably stable across the full range of proportions. The results confirm the ability of the CES model to correct for bias with respect to the sizes of standard list, in particular for those variants in which the significance is maximized over a range of subcache sizes.

A final experiment was conducted to determine the influence of the choice of k on the recall rate of top-k active cache queries, when the lengths of the cache standard lists is variable. In this test, the list lengths were selected uniformly at random in the range 1 to 100. The cache size was again chosen to be 25%. Figure 5.26 shows a peak in performance for all methods with k between 30 and 40, with the *MaxCorr* and *MaxRatio* methods again offering the greatest stability across the range.

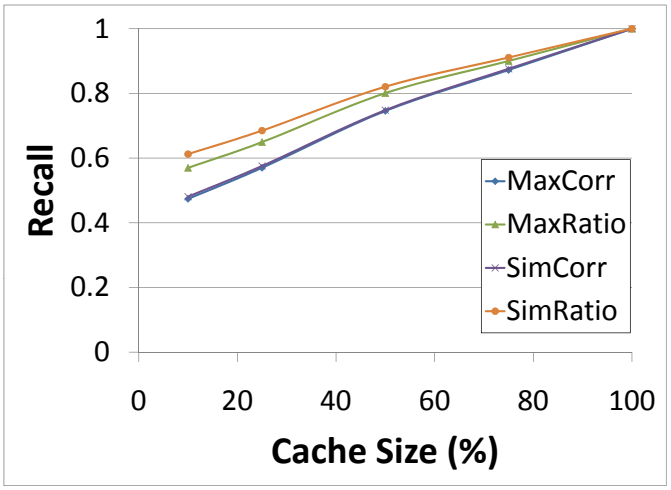

Figure 5.13 Average cache recall for active caching hits for the MovieLens dataset, taken across a range of cache sizes with $k = \lambda = 10$.

5.6 Summary

This chapter proposes an improvement to the original partial order based Cached Estimation (CES) model for the estimation of top-k query results using cached information. The model is based on shared-neighbor similarity measures that assess the statistical significance of the relationship between objects based on their shared neighborhood. The main contribution of the this improvement is to facilitate the design of shared-neighbor ranking formulae for active caching that allow for variation of (and comparison across) such parameters as the size of the cache, the length of ranked lists stored in the cache, and the number of items requested by the query. The experimental results of the previous section indicate that the performance of the CES-derived *MaxRatio* and *MaxCorr* ranking functions is somewhat more stable than their counterparts *SimRatio* and *SimCorr* when cached list sizes were allowed to vary.

Shared-neighbor ranking formulae do not make use of monotonicity like partial order based approach presented in the last chapter. Test results showed significant improvement in recall by using shared-neighbor approach. The very strong overall performance in terms of hit rate, recall and execution cost of the shared-neighbor ranking formulae for active caching provides an intriguing answer to the question of cache management for databases.

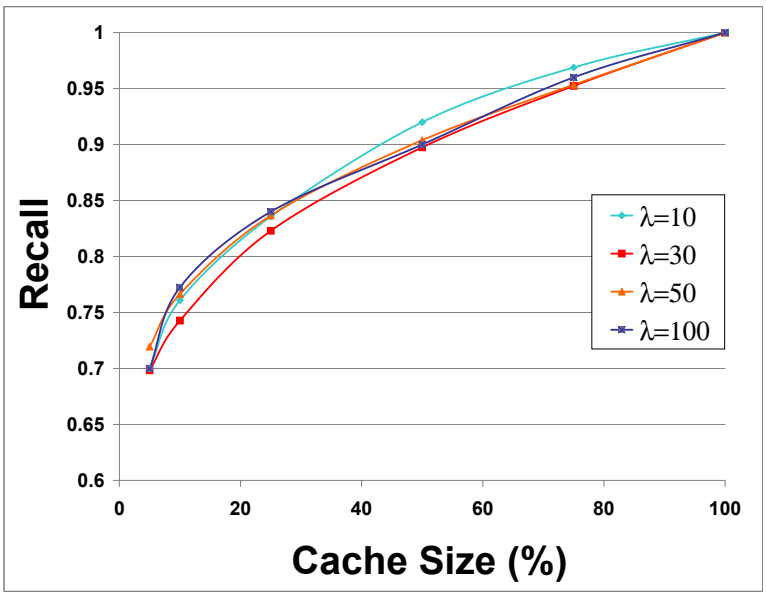

Figure 5.14 Average cache recall for top-k active caching hits using the *SimRatio* measure, across a range of cache sizes and list sizes $\lambda = k$.

Whereas the conventional approach is to fill the cache with those items most likely to be requested in future queries, the experimental results show that shared-neighbor active caching in which the cache is selected so as to provide uniform coverage of the data set from which most if not all query results are actively generated. For some applications, it may even suffice to answer all queries actively without ever referring to the original data.

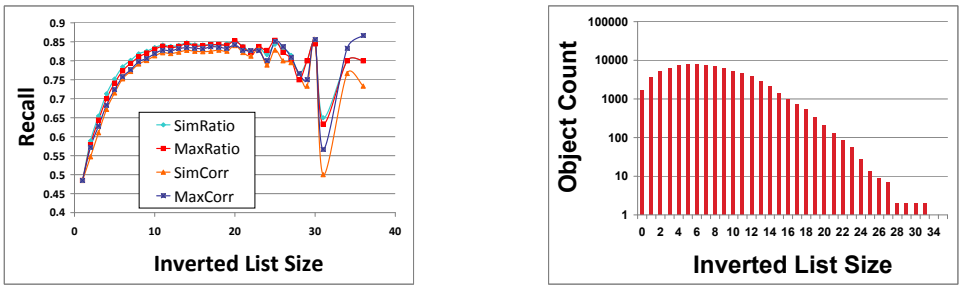

Figure 5.15 Average estimation recall values for active cache hits as a function of query inverted list size, for the ALOI dataset with cache size 25% and $k = \lambda = 30$. The histogram shows the numbers of query items with inverted lists of a given size. The high variation in recall for large inverted lists is due to the very small number of instances of these lists.

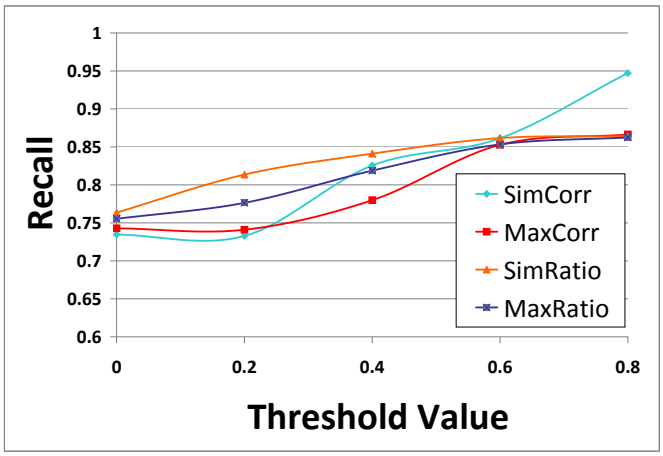

Figure 5.16 Average estimation recall rates of top-30 active cache hits for the ALOI dataset, in which all items in the result list satisfy minimum thresholds on similarity measure values.

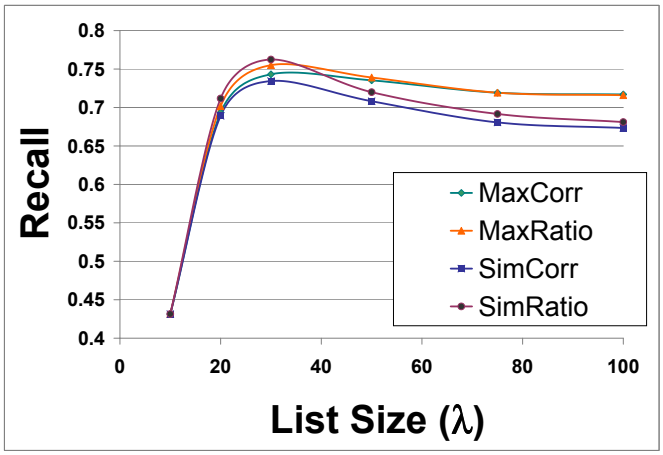

Figure 5.17 The effect of varying cache list size λ on the average estimation recall rates of top-30 similarity query cache hits, for the ALOI dataset with a cache size of 25%.

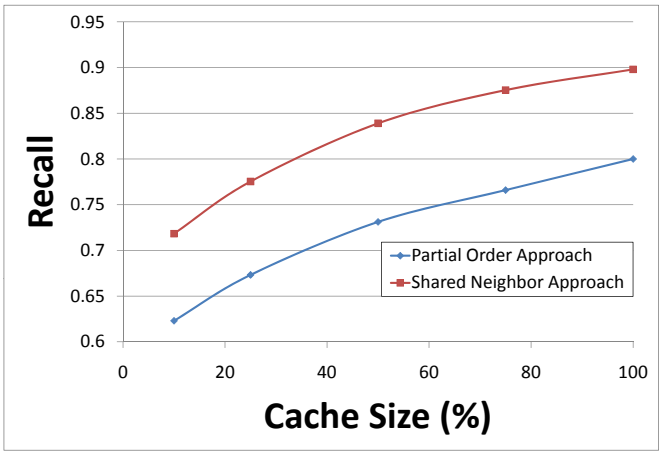

Figure 5.18 Average cache recall for active caching hits for the ALOI dataset, comparison between partial order approach and shared neighbor approach using $k = \lambda = 30$.

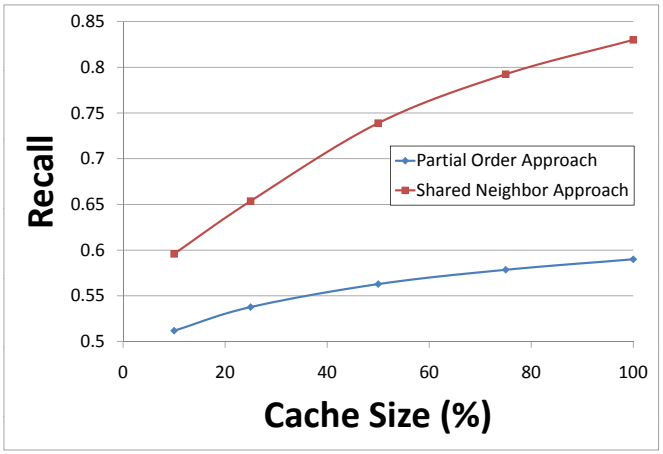

Figure 5.19 Average cache recall for active caching hits for the Reuters dataset, comparison between partial order approach and shared neighbor approach using $k = \lambda = 30$.

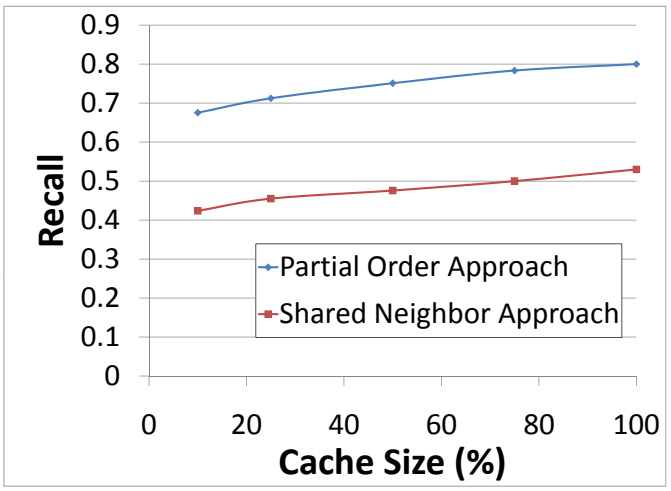

Figure 5.20 Average cache recall for active caching hits for the KDD dataset, comparison between partial order approach and shared neighbor approach using $k = \lambda = 20$.

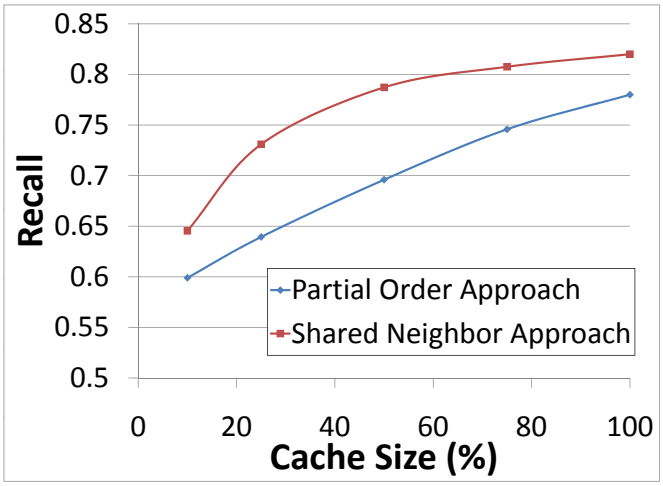

Figure 5.21 Average cache recall for active caching hits for the Cover Type dataset, comparison between partial order approach and shared neighbor approach using $k = \lambda = 20$.

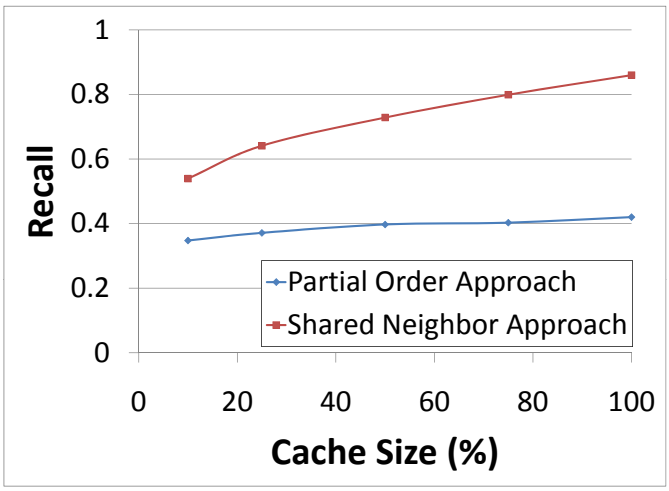

Figure 5.22 Average cache recall for active caching hits for the Jester dataset, comparison between partial order approach and shared neighbor approach using $k = \lambda = 20$.

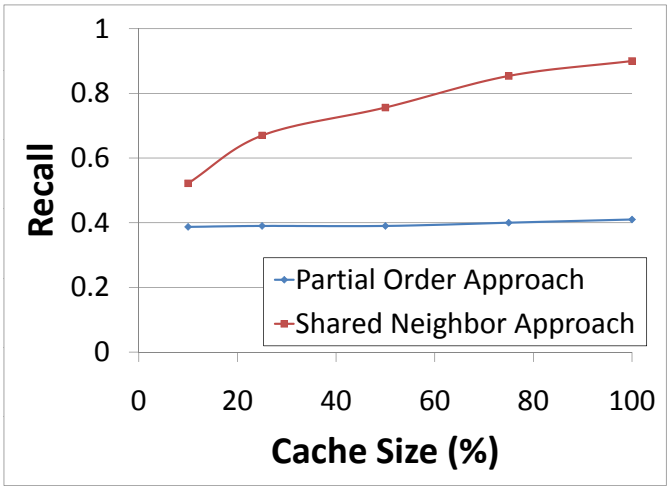

Figure 5.23 Average cache recall for active caching hits for the MovieLens dataset, comparison between partial order approach and shared neighbor approach using $k = \lambda = 10$.

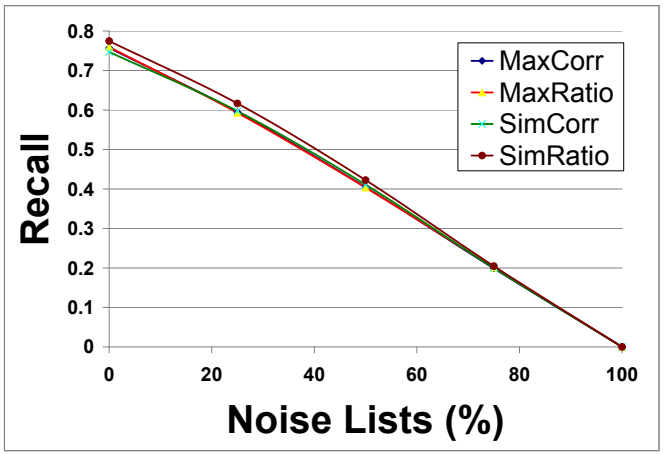

Figure 5.24 Average estimation recall rates for ALOI active cache hits, with a cache size of 25% and with $k = \lambda = 30$, plotted against the proportion of noise lists.

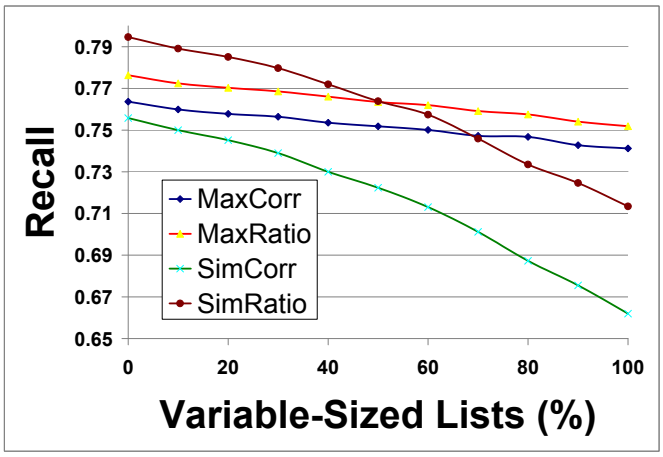

Figure 5.25 Average estimation recall rates of top-30 active cache hits for the ALOI dataset, plotted for a cache size of 25% against various proportions of standard lists having lengths ranging between 1 and 100, with the remainder of the lists having length 30.

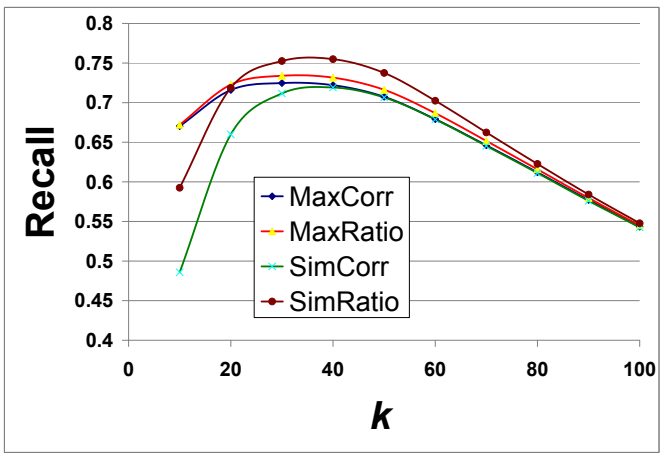

Figure 5.26 Average estimation recall rates of top-k active cache hits for the ALOI dataset, plotted for a cache size of 25% against different values of k, with standard list sizes randomly selected between 1 and 100.

CHAPTER 6

GREEDY BALANCING CACHE SELECTION POLICY

6.1 Introduction

In many databases and Web applications, caching is often employed to improve response time and reduce the server workload. A cache is a temporary storage area where data can be stored for quick access. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or recomputing the original data, so that the average access time is shorter. Caching can improve the performance of an application by reducing access latency, server load and network traffic. Caching implemented as proxies between users and web servers reduces server load, network bandwidth usage as well as user access latency [37, 85, 111]. Normally only a small proportion of the entire dataset is cached, and the criteria by which cache items are selected is crucial to the performance of any caching solution. This selection can be performed in many ways; however, the ultimate goal of selection is to maximize the number of queries that can be answered from the cache.

Several techniques have been proposed in the research literature to select the most appropriate data for caching. Typically, the content of the cache is dynamically updated in order to adapt to changes in user request patterns. Insertion of new items into the cache first requires that items be selected for replacement. Most cache replacement strategies select for deletion either the least recently used cache element (LRU) or the least frequently used element (LFU). Both the LFU and LRU cache replacement strategies take into account the popularity of the data with respect to query requests. The LRU approach can be viewed as a form of temporal locality, whereas the LFU approach can be viewed as a form of spatial locality in that it preserves cache objects residing in areas where the query distribution is dense. Although traditional caching strategies allow for dynamic updates, researchers

have also considered the problem of selecting a static cache so as to be able to answer the maximum number of queries for given distributions of data and queries [130].

Caching strategies can be divided into two broad classes, 'passive' and 'active'. With passive caching, if the query result cannot be retrieved from the main memory cache, the result is constructed from information residing in secondary storage. Active caching extends the performance of a passive cache so that whenever the target result is not explicitly available in the cache, it makes use of the stored results from previous queries to estimate the result for the current query. This estimation can be viewed as a form of approximate query processing [83]. Active caching with some query processing capability can significantly reduce the server workload and improve the performance and scalability [80]. As with traditional caching, the performance of any active caching solution depends greatly on the selection of cache data. Since the size of cache memory is much smaller than the total data size, it is essential to select data that can help answer the maximum number of queries. Active caching research in the past has not addressed the issue of appropriate data selection for the active caches.

For the problem of top-k similarity search, recently-introduced active cache strategies have been shown to significantly improve the query hit rate [49, 105]. The Cache-Estimated Significance (CES) model [49] utilizes the spatial locality amongst the cached content to process non-cached queries. The result of a non-cached query is estimated based on the relationship of data objects with the cached top-k neighbor lists: if a given database object s and the query object q both appear in the same cached lists, then the likelihood is high that s belongs to the top-k result for q. Inverted lists associated with the cached top-k result list are also maintained in the cache to help estimate the similarity between the query object and other objects in the database.

This chapter focuses on the problem of selecting a set of cache objects so as to allow CES-style query result estimation to be reliably performed on any object in the database. The ideal selection strategy for CES would allow all objects of the database to have equal

representation in the cached top-k neighbor lists. If all data items were to have equal representation in the cached top-k lists, the lengths of the corresponding inverted top-k lists must be balanced. The main contribution of this chapter is an efficient greedy cache selection strategy that achieves an balancing of inverted neighbor lists that greatly improves the effectiveness of CES.

The remainder of the chapter is organized as follows. Section 6.2 gives background information on traditional passive cache management techniques, and argue that these techniques are not appropriate for active caching. Section 6.3 describes the proposed greedy balancing strategy. Section 6.4, provides and discusses experimental results.

6.2 Background

This section first gives an overview of issues related to caching, before focusing on the Cache-Estimated Significance model that serves as the basis.

6.2.1 Caching Strategies

Most existing passive caching techniques have been proposed either for relational databases or the Web, and most allow for dynamic updates of the cache so as to adapt to changes in user access patterns. As mentioned earlier, LRU and LFU (along with their variants) are the most commonly used cache replacement strategies. The effects of these strategies, and the motivations for their use, can be understood in terms of the temporal and spatial locality of data access patterns.

The locality of reference principle dictates that an application does not access all of its data at once with equal probability, but instead exhibits dependencies on the temporal and/or spatial properties of the data. The temporal locality property suggests that if a data item is requested at a given time, then there is a high likelihood of it being requested again in the near future [14]. Markatos compared caching of the most popular queries with caching of the most recently accessed queries and showed that (spatially-based) LFU

is advantageous for small caches, while (temporally-based) LRU has significantly better performance for large caches [86]. In another study of database access traces, the proportion of repeated queries was found to be roughly 30% to 40%, with the majority of repeated queries occurring within a short time interval after the intial query [139]. In one of the traces (Vivisimo), about 65% of query repetitions occurred within a one-hour interval [139]. Rizzo and Vicisano [110] and Cao and Irani [21] also observed this property in web proxy server traces.

The LFU replacement policy is typically used when the data is known to follow the Zipf distribution. Zipf's law assumes that the relative probability of a request for the i-th most popular data item is proportional to $1/i$. Breslau et al. [18] examined six traces from proxy servers at academic institutions, corporations and ISPs and found that the distribution of page requests generally followed a Zipf-like distribution. Serpanos and Wolf [123] verified that high hit rates can be achieved for web page caching using strategies based on the Zipf distribution. Markatos discovered a large number of frequently-posed queries in the retrieval logs of search engines that constitute excellent candidates for caching [86]; in general, the query frequencies follow a Zipf distribution [139, 40, 32].

Along with temporal locality, the spatial locality principle also dictates the pattern of usage particularly in memory caches. The spatial locality principle states that if a given data item is requested, then there is a high likelihood of similar data items being requested in the near future [14]. Spatial locality has been effectively used in computer memory caches, where recently referenced data is cached together with similar but less-recently referenced data. Spatial locality, when used effectively, alleviates the latency and bandwidth issues of computer memory by boosting the effect of prefetching [44]. I/O scheduling and prefetching can effectively exploit spatial locality and dramatically improve disk throughput [34]. Kampe & Dahlgren focused on the characteristics of locality in terms of spatial and temporal proximity, and presented a scheme to exploit this locality for cache management [57]. Ding, Jiang and Chen proposed the DULO buffer cache management

scheme, which exploits both temporal and spatial locality [34]. Sequeira *et al.* proposed an approach for increasing the spatial locality of programs for data mining, decreasing the average working set size and thereby decreasing the page fault rate [122].

Cache management policies based on spatial and temporal locality have been widely studied in the context of passive caching, but essentially no research has yet been conducted on the problem of selection strategies for active caching. With passive caching, replacement strategies are necessary due to the small coverage of the query range provided by the cache. Active caching strategies seek to provide good coverage of the full query range without relying on cache updates to anticipate query access patterns. As such, temporal locality-based cache selection strategies such as LRU are not appropriate when full coverage is sought; spatial locality-based selection strategies such as LFU are also not appropriate when applied relative to only a small subset of the dataset or query range. In order to achieve good coverage of the dataset, recent active caching strategies proposed for top-k similarity queries used random sampling to populate the cache [49, 105], achieving hit rate and average recall performance significantly better than that which would have been achieved using traditional passive caching. Their results suggest that effective estimation of query results may be performed without ever referring to the original data on disk [49].

6.2.2 The Cache-Estimated Significance Model

The Cache-Estimated Significance (CES) model [49] estimates the results of top-k similarity queries whose result is not readily available in the cache. In this model, similarity measures based on shared-neighbor information were introduced which assess the strength of the relationship between two objects as a function of the number of cached top-k neighbor lists that contain both objects. The cached objects thereby constitute a set of potential witnesses to the relationship between two objects. The main contribution of the CES model is the facilitation the design of shared-neighbor ranking formulae for active caching. These ranking functions can correct for bias relating to variations in such quantities as the size of

the cache, the length of ranked lists stored in the cache, and the number of items requested by the query, all without any knowledge of the actual similarity values.

To support this approach, a cache structure was introduced to retain neighbor lists (the *relevant* sets) and to maintain their associated inverted lists (the *inverted relevant* sets). Given a dataset S drawn from domain D, and a list length $\lambda > 0$, for any object $v \in S$, the relevant set of length λ based at v is represented by $\mathrm{Q}(v, \lambda)$. Taken together over all $v \in S$, the relevant sets induce a collection of inverted relevant sets $\mathrm{Q}^{-1}(u, \lambda) = \{v \in S : u \in \mathrm{Q}(v, \lambda)\}$ for every choice of $u \in S$. For a given $u \in S$, inverted relevant sets for u can also be defined with respect to the cache $\mathcal{C}(C, \lambda)$, by restricting the membership of the lists to objects of C instead of S, as follows:

$$\mathrm{Q}_C^{-1}(u, \lambda) \stackrel{\triangle}{=} \mathrm{Q}^{-1}(u, \lambda) \cap C = \{v \in C : u \in \mathrm{Q}(v, \lambda)\}.$$

The collection of all such inverted lists taken over all choices of $u \in S$ is referred to as the *inverted cache* corresponding to $\mathcal{C}(C, \lambda)$, and can be denoted by $\mathcal{C}^{-1}(C, \lambda) \stackrel{\triangle}{=} \{\mathrm{Q}_C^{-1}(v, \lambda) : v \in C\}$. To estimate the top-$k$ relevant set $\mathrm{Q}(v, k)$ for some object $v \notin C$, the cached object set C would constitute a set of potential *witnesses* to the relationship between two objects v and w in S.

With respect to any object $v \in S$, the CES model allows the rank of another object w to be estimated. The relationships between $\mathrm{Q}(w, \lambda)$ and $\mathrm{Q}(v, \lambda)$ on the one hand, and $\mathrm{Q}^{-1}(w, \lambda)$ and $\mathrm{Q}^{-1}(v, \lambda)$ on the other, serves as the foundation of a *rank measure* of the similarity between w and v. Each object of $\mathrm{Q}(x, \lambda) \cap \mathrm{Q}(v, \lambda)$ would support the contention that w and v were similar, and each object of $\mathrm{Q}(w, \lambda) \setminus \mathrm{Q}(v, \lambda)$ or $\mathrm{Q}(v, \lambda) \setminus \mathrm{Q}(w, \lambda)$ would work against it. The same would be true for inverted sets, with each of $\mathrm{Q}^{-1}(w, \lambda) \cap \mathrm{Q}^{-1}(v, \lambda)$ testifying as to the similarity of w and v, and each object of $\mathrm{Q}^{-1}(x, \lambda) \setminus \mathrm{Q}^{-1}(v, \lambda)$ or $\mathrm{Q}^{-1}(v, \lambda) \setminus \mathrm{Q}^{-1}(w, \lambda)$ refuting it.

Houle, Oria and Qasim [49] introduced and evaluated six ranking functions under the CES model. As a representative of these measures, one of the simplest measures, *SimInt*,

which performed well in their experimentation. Given two items v and w, *SimInt* is defined as the size of the intersection of the cache inverted neighbor lists associated with v and w:

$$\textit{SimInt}_{C,\lambda}(v,w) \stackrel{\triangle}{=} |\mathrm{Q}_C^{-1}(v,\lambda) \cap \mathrm{Q}_C^{-1}(w,\lambda)|.$$

Their proposed active caching approach showed very strong overall performance with *SimInt* tested as being the most accurate ranking function in many situations. This is equally applicable to any CES-based ranking function in general, and the six functions proposed in [49] in particular. More details on CES-based ranking functions can be found in [49].

6.3 Greedy Balancing Strategy

The performance of the CES active strategy has been shown to depend on the length of the cached inverted list associated with a given query object q [49]. Generally speaking, the quality of the query result estimate degrades as the length of the query inverted list is reduced — the most extreme case occurs when the list is empty, in which case no result can be estimated. This suggests that for providing consistently good estimates for arbitrary similarity queries, cache selection strategies should seek to achieve the best possible balance in terms of the inverted list lengths. To be effective, any such strategy would need to provide good coverage for all possible queries, be easy to implement, and be efficient to compute.

One way of assessing the degree of balance of inverted cache lists is through the variance of the lengths of these lists, with low variance indicating a high degree of balance. Let $\gamma(s_i)$ denote the length of the inverted list associated with a given object $s_i \in S$ in $\mathcal{C}^{-1}(C,\lambda)$. The goal is to select C so as to minimize the variance of the lengths of these lists, which can be expressed as

$$\begin{aligned}\sigma^2(C,\lambda) &= \frac{1}{n}\sum_{i=1}^{n}\left(\gamma(s_i)-\mu(C,\lambda)\right)^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}\gamma^2(s_i)-\mu^2(C,\lambda),\end{aligned}$$

where $\mu(C, \lambda) = \frac{\lambda \cdot |C|}{n}$ is the average length of the inverted cache lists.

With the objective of achieving better inverted list balance than is possible using uniform random cache selection, a greedy selection strategy for minimizing the variance of inverted cache list lengths is proposed. Objects are introduced into the cache one by one; at each step, an object is chosen so that the increase in the inverted list length variance is the minimum possible (or equivalently, the decrease in the variance is the maximum possible). Determining the next object for inclusion in the cache, if performed in a straightforward manner, would be prohibitively expensive for large databases: computing $\sigma^2(C_t \cup \{s\}, \lambda)$ for every candidate $s \in S \setminus C_t$ for the partial cache $\mathcal{C}(C_t, \lambda)$ at every insertion step $1 \leq t \leq m$ would require at least $\Omega(mn^2)$ operations overall, where $m = |C|$ is the number of objects in the final cache C.

This section shows that the greedy list variance minimization can be performed much more efficiently. Let C_t be an existing set of cache objects selected as of the current iteration of greedy selection. For every $s \in S$, maintain a score equal to the total size of the inverted lists associated with the objects in the relevant set $\mathrm{Q}(s, \lambda)$. More precisely, the score is defined to be

$$\Gamma_t(s) = \sum_{v \in \mathrm{Q}(s,\lambda)} \gamma_t(v),$$

where $\gamma_t(v)$ is the length of the inverted list associated with $v \in S$ in $\mathcal{C}^{-1}(C_t, \lambda)$. For example, the score of object 6 in Figure 6.1 equals 3 — the sum of the inverted list lengths of its neighbor objects 6, 0, and 5.

The following lemma implies that the greedy selection process reduces to determining an object in $S \setminus C$ for which Γ is minimized.

Lemma 6.3.1. *Let $\mathcal{C}(C_t, \lambda)$ be a cache for some subset $C_t \subset S$. Let s_* be any object minimizing $\Gamma_t(s)$ over all choices of objects of $S \setminus C_t$. Then*

$$\min_{s \in S \setminus C} \sigma^2(C_t \cup \{s\}, \lambda) = \sigma^2(C_t \cup \{s_*\}, \lambda).$$

Proof. Let s be any object of $S \setminus C_t$, and let $t = |C_t|$. Consider now the set of cache objects $C_{t+1} = C_t \cup \{s\}$ obtained by augmenting C_t with s. Let $\gamma_{t+1}(s_i)$ be the length of the inverted cache list for object s_i after the insertion of s into the cache. Applying (6.1), and noting that the average inverted list length grows from $\frac{t\lambda}{n}$ to $\frac{(t+1)\lambda}{n}$, the resulting change in variance is then

$$\begin{aligned}\Delta &= \sigma^2(C_{t+1}, \lambda) - \sigma^2(C_t, \lambda) \\ &= \frac{1}{n}\sum_{i=1} n\left(\gamma_{t+1}^2(s_i) - \gamma_t^2(s_i)\right) - \frac{\lambda^2}{n^2}(2t+1)\,.\end{aligned}$$

Note that the set I of objects whose inverted lists change as a result of the introduction of s into the cache is precisely $I = \mathbf{Q}(s, \lambda)$, and that each of the affected inverted lists grows by exactly one element. Simplifying,

$$\begin{aligned}\Delta &= \frac{1}{n}\sum_{s_i \in I}\left((\gamma_t(s_i)+1)^2 - \gamma_t^2(s_i)\right) - \frac{\lambda^2}{n^2}(2t+1) \\ &= \frac{2}{n}\sum_{s_i \in I}\gamma_t(s_i) + \frac{\lambda}{n} - \frac{\lambda^2}{n^2}(2t+1)\,.\end{aligned}$$

In this expression for Δ, only the summation depends on the choice of object s. Therefore minimizing Δ over all choices of $s \in S \setminus C_t$ is equivalent to minimizing

$$\Gamma_t(s) = \sum_{v \in \mathbf{Q}(s,\lambda)} \gamma_t(v).$$

The result then follows. □

Maintaining the list length sums $\Gamma_t(s)$ for all remaining candidate objects $s \in S \setminus C_t$ can be managed much more efficiently than the direct recomputation of variances. Below are the details as to how the greedy cache balancing strategy can be implemented using list length sums, as justified by Lemma 6.3.1. Initially, at step $t = 0$ assume that the cache set C_0 is empty, and that λ-nearest neighbor lists are available for each object in S. The goal is to determine a set of objects $C = C_m$ with specified cardinality m whose inverted neighbor list lengths have the smallest possible variance.

In the description of the algorithm, the following structures are maintained:

- For a specified subset of $S \setminus C_t$, a set of objects $HoldingSet$ together with their identifiers and a linked list to inverted neighbors. For example, object 6 in the $HoldingSet$ shown in Figure 6.1 heads a linked list to objects 4 and 5, indicating that objects 6, 4 and 5 are the 3-nearest neighbors of object 6.

- A list $RankedList$ of objects achieving a common value of Γ from among the objects currently available for selection. Each object $s_i \in RankedList$ is associated with an identifier $i \in [1, n]$, its inverted list length $\gamma(s_i)$, and its neighbor set $\mathrm{Q}(s_i, \lambda)$.

- A min-heap data structure $ScoreHeap$ storing pointers to all objects not yet selected for inclusion in the cache. The min-heap is organized according to the Γ scores of the objects it stores. All objects sharing the same Γ value are stored in a common $RankedList$ structure linked to one node of the min-heap. An example is shown in Figure 6.1.

At each selection step, The greedy strategy performs the following operations. An object s is randomly chosen from the $RankedList$ referenced by the top element of $ScoreHeap$, and a new cache object set $C_{t+1} = C_t \cup \{s\}$ is formed. As a result of the selection of s, the inverted list lengths of all neighbors of s (including s itself) are incremented by one. This incrementation necessitates an update of the Γ values of each of the objects of $S \setminus C_t$ that share at least one neighbor with s. Any objects whose Γ value increases would be reassigned to the $RankedList$ structure associated with its new value.

The objects requiring an update of their Γ values (and relocation to another $RankedList$ structure) can be characterized as follows: for each object v in $\mathrm{Q}(s, \lambda)$, an update is required for each object of $\mathrm{Q}^{-1}(v, \lambda)$. As these objects are discovered, they are inserted into a $HoldingSet$ structure; once all candidates have been inserted, the scores are updated, and objects are relocated to other $RankedList$ structures. If no $RankedList$ structure exists for

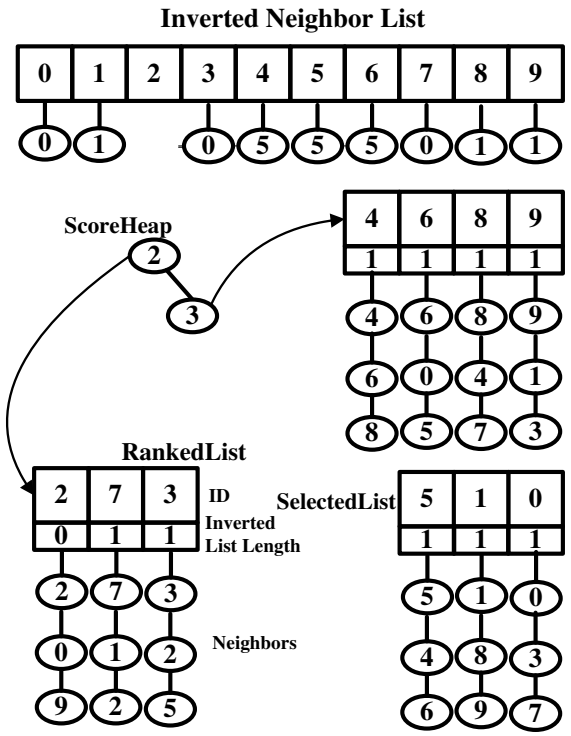

Figure 6.1 Data structures of the GreedyBalance algorithm for a set of 10 objects after the objects 0, 1, and 5 have been selected.

a given score, a new $RankedList$ is created and inserted into $ScoreHeap$. A pseudocode description of the greedy selection process is provided below (Algorithm 1).

Figure 6.2 shows an example state of the structures for the neighbor lists of Figure 6.1, where the object selected from the $RankedList$ with score 0 is object 2. As a result of the selection, the inverted list lengths of neighbor objects 2, 0 and 9 are incremented by one. The set of objects containing at least one of $\{0, 2, 9\}$ as neighbors is $\{3, 6, 7, 9\}$. Before the selection, objects 3 and 7 have score 2, and objects 6 and 9 have score 3. After the update, object 2 is added to the cache. Objects 3 and 7 are moved to the $RankedList$ structure for score 3, resulting in the list for score 2 becoming empty. In the case of objects 6 and 9, a new $RankedList$ structure for score value 4 is created to store it, and is inserted into the min-heap.

The GreedyBalance heuristic allows for much faster cache selection as compared to a direct computation of list length variances, in that only a limited number of list objects need

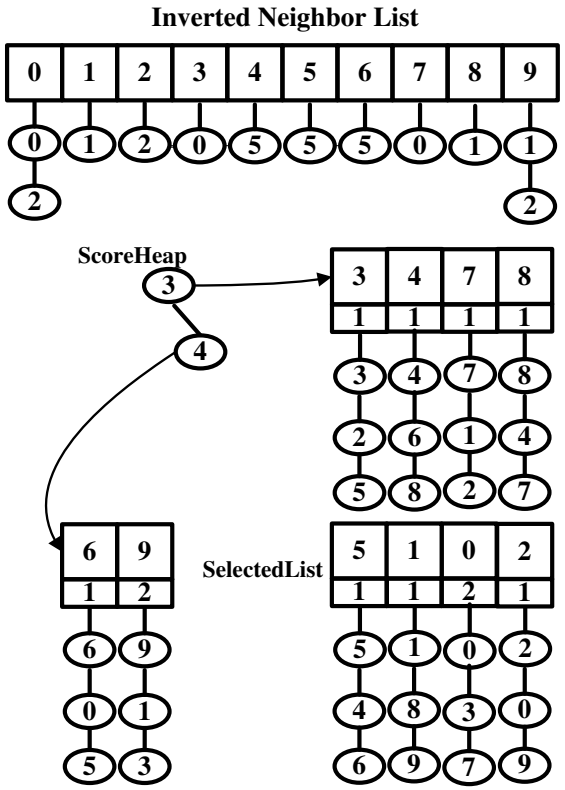

Figure 6.2 State of each data structure after the object 2 is selected.

be examined at every selection. These objects are precisely those whose cached neighbor lists contain one of the λ neighbors of the object entering the cache. Evaluation of the balancing heuristic focuses on the degree of balance of inverted list lengths achieved using the heuristic, as well as the impact of balancing on the performance of CES active caching.

6.4 Evaluation

The impact of inverted list balancing strategy is assessed by implementing CES together with GreedyBalance using the *SimInt* measure for inverted list similarity. *SimInt* was selected as it was shown to perform well across the different datasets considered in [49]. This variant, referred as CES-GB, was tested against basic CES, again implemented using *SimInt*.

6.4.1 Performance Measures

The query processing performances of CES and CES-GB were assessed in terms of two measures: the *hit rate* and *recall*. Given a schedule of queries, the *hit rate* is traditionally defined to be the proportion of queries for which the result is explicitly resident in the main-memory cache. Here, the definition is extented to include those cases where a query result can be estimated using active caching. A miss occurs when active caching does not lead to an estimate of the query result due to the query object having no assocation with a cached neighbor list or a cached inverted list.

Consider now the item set retrieved by any given top-k query operating on the cache. The *recall* of the query is defined as the proportion of this result that would also appear in the result of the same query when applied to the full database. Since the cache query result and the database query result are of equal size, this definition of recall is equivalent to that of query *precision*, and the terms can be used interchangeably. In order to better contrast the estimation powers of CES and CES-GB, experimental evaluation uses active caching to produce an estimated query result regardless of whether or not the result is explicitly

stored in the cache. Accordingly, query recall values may be less than 1 even when the top-k result list is explicitly stored in the cache. When a cache miss occurs, as the query result cannot be estimated using the cache, the recall of the query is considered to be 0.

6.4.2 Hit Rate

Using the datasets described in Chapter 3, several top-k query-by-example retrieval experiments were conducted, over a variety of cache and neighbor list sizes, and choices of k. For each experiment, the cache objects provided to CES were selected uniformly at random from among the set of available objects, whereas for CES-GB the GreedyBalance heuristic was employed. The cached information consisted of the standard neighbor list for each cache item, and well as their associated inverted lists. After setting up the cache, each of the items of the full collection served as the basis of a query-by-example operation.

Tests of CES-GB generally showed higher hit rates as compared with standard CES, for cache proportions ranging between 5% and 50% (see Figures 6.3, 6.4, 6.5 , 6.6, 6.7 and 6.8). The differences in hit rates are very significant for smaller caches, with the advantage held by CES-GB diminishing as the cache size increases. For the ALOI, KDDCup and CoverType datasets with a cache proportion of 5%, CES-GB was able to estimate a result for more than 85% of the queries, compared to just over 60% for CES. For the RCV1 dataset, the hit rates for both methods were lower, although CES-GB still outperformed CES by a margin of more than 20%. In practice, usually only a very small proportion of a dataset is kept in the cache.

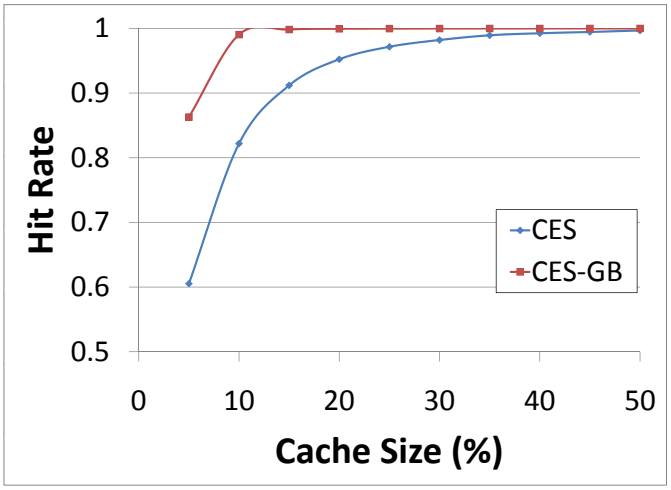

Figure 6.3 Hit rate for the ALOI dataset for cache proportions of between 5% and 50%, with neighbor list size $\lambda = 20$.

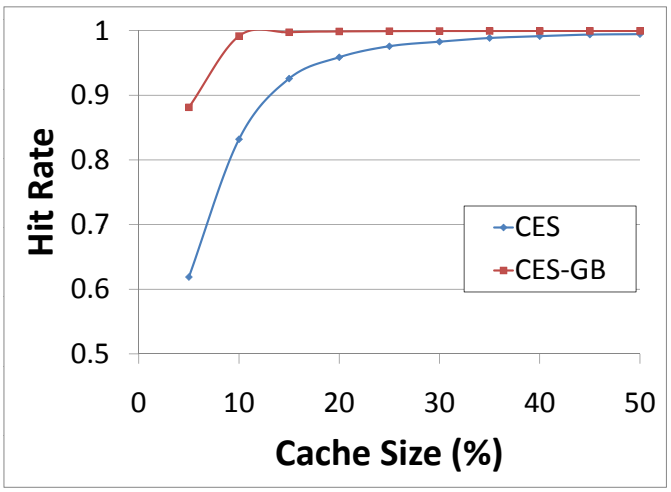

Figure 6.4 Hit rate for the KDDCup dataset for cache proportions of between 5% and 50%, with neighbor list size $\lambda = 20$.

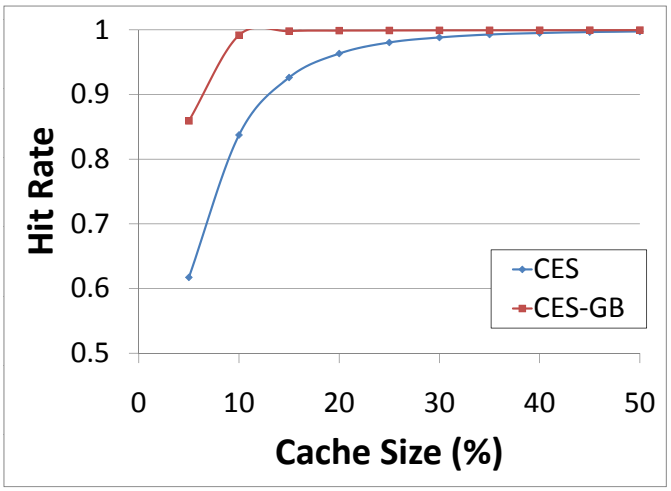

Figure 6.5 Hit rate for the CoverType dataset for cache proportions of between 5% and 50%, with neighbor list size $\lambda = 20$.

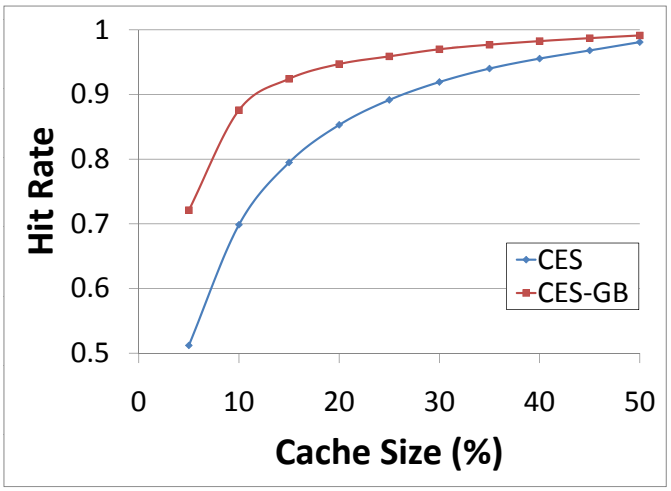

Figure 6.6 Hit rate for the RCV1 dataset for cache proportions of between 5% and 50%, with neighbor list size $\lambda = 20$.

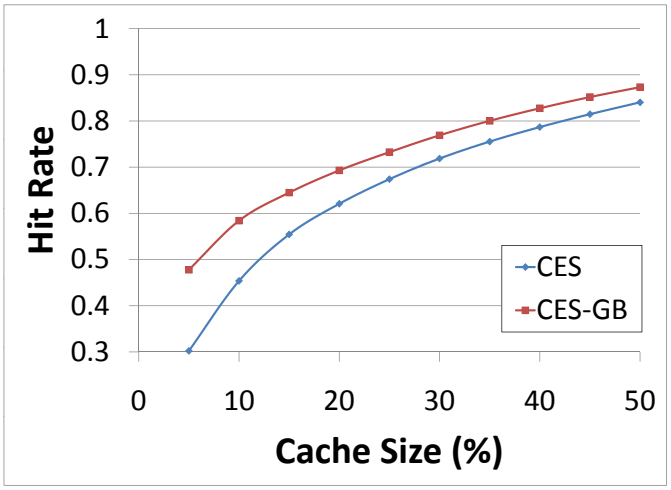

Figure 6.7 Hit rate for the Jester dataset for cache proportions of between 5% and 50%, with neighbor list size $\lambda = 20$.

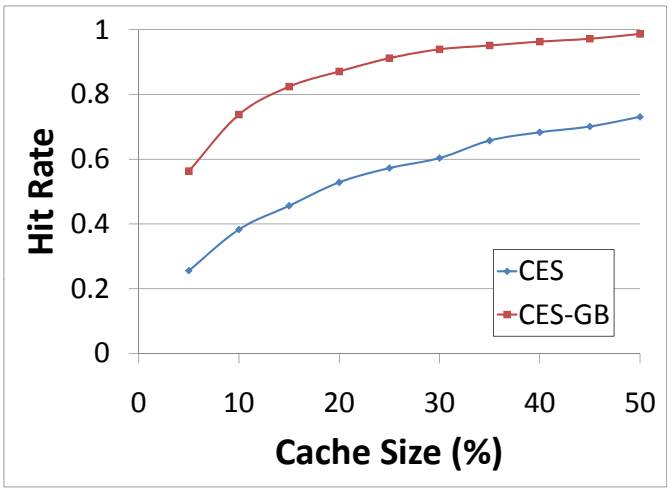

Figure 6.8 Hit rate for the MovieLens dataset for cache proportions of between 5% and 50%, with neighbor list size $\lambda = 20$.

6.4.3 Average Recall

The experiments to measure average recall values were conducted for the same choices of cache sizes and list sizes as for the hit rate; the results are shown in Figures 6.9, 6.10,

6.11, 6.12 and 6.13. For cache sizes of size 10% or more, the average inverted neighbor list length was 2 or more (due to the choice of $\lambda = 20$), allowing both CES-GB and CES to achieve very high average recall rates. For the smaller cache sizes, CES-GB substantially outperformed CES, while for cache proportions in excess of 30%, CES held only a very slight advantage over CES-GB. When the cache size was set to 5%, the average inverted list length was only 1, which led to a large drop in average recall values for both methods. Nevertheless, for the ALOI, KDDCup and CoverType datasets, CES-GB achieved average recall rates in excess of 50% even despite the small average inverted list length, even with the imposed limitation of using an estimate when the result was present in the cache, and even with the recall of cache misses treated as zero. These rates were more than 10% higher than those for CES. For the RCV1 set, the average recall of CES-GB was slightly over 30%, with CES achieving 25%.

The results for average recall together with those for hit rates show that very effective performance can be achieved while only rarely needing to access information on disk.

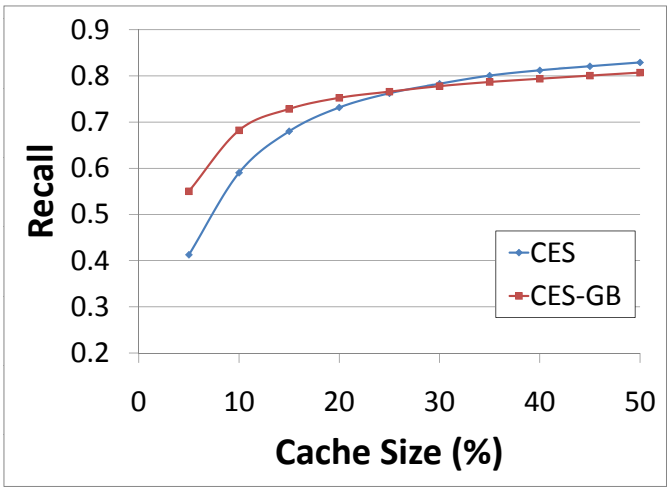

Figure 6.9 Average recall for top-k queries using the ALOI dataset with list size $\lambda = k = 20$.

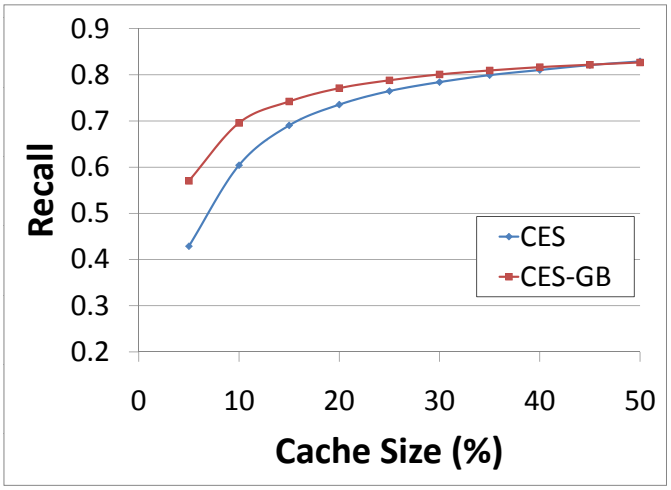

Figure 6.10 Average recall for top-k queries using the KDDCup dataset with list size $\lambda = k = 20$.

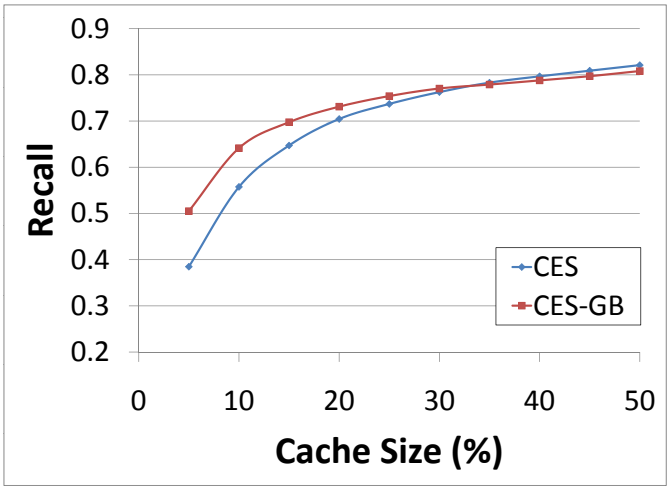

Figure 6.11 Average recall for top-k queries using the CoverType dataset with list size $\lambda = k = 20$.

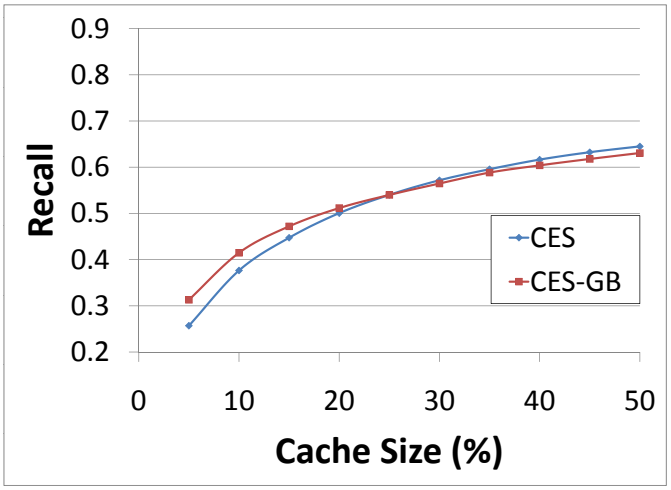

Figure 6.12 Average recall for top-k queries using the RCV1 dataset with list size $\lambda = k = 20$.

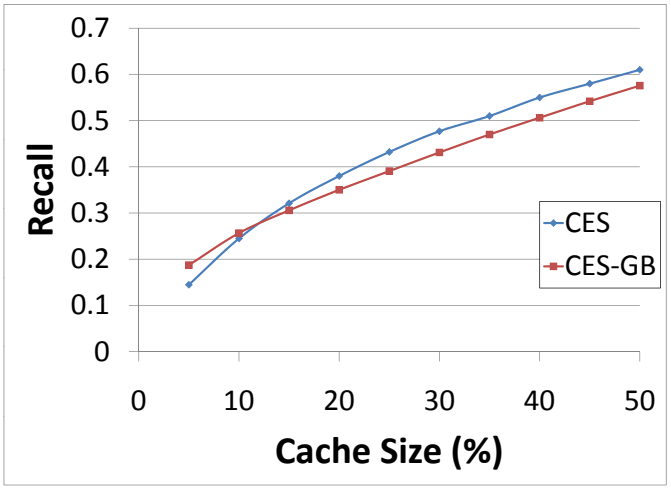

Figure 6.13 Average recall for top-k queries using the Jester dataset with list size $\lambda = k = 20$.

6.4.4 Inverted Neighbor List Balancing

The performance of the CES active strategy strongly depends on the length of the cached inverted list associated with a given query object q [49]. The proposed solution is targeted

to achieve the best possible balance in terms of the inverted list lengths. The degree of balance of inverted cache lists can be assessed through the variance of the lengths of these lists, with low variance indicating a high degree of balance. To be effective, this strategy should also provide good coverage for all possible queries.

In Figures 6.14, 6.15, 6.16, 6.17, 6.18, and 6.19, histogram plots are provided showing the numbers of query items against inverted list length for both CES-GB and CES, with the query proportion set at 20% and neighbor list lengths chosen as size $\lambda = 20$. For all examples, the mean inverted neighbor list length is 4. The results show far less variation in the inverted list lengths for CES-GB as compared against CES. Query coverage is dramatically improved using the greedy selection strategy, as evidenced by the much smaller numbers of inverted lists with length zero.

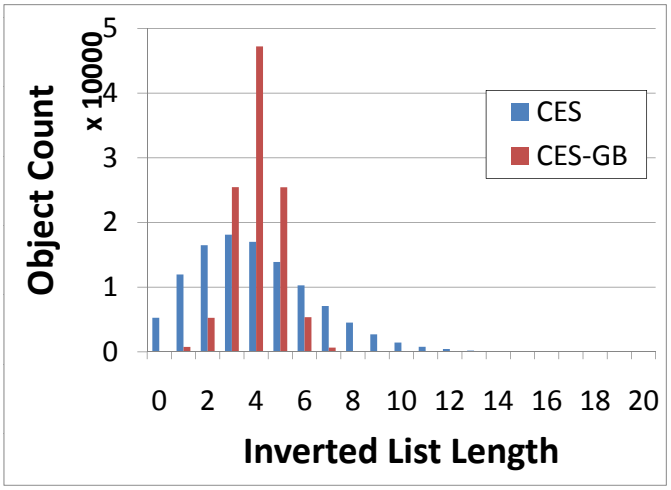

Figure 6.14 Histograms of object count versus inverted neighbor list length for the ALOI dataset, with neighbor list size $\lambda = 20$ and a cache proportion of 20%.

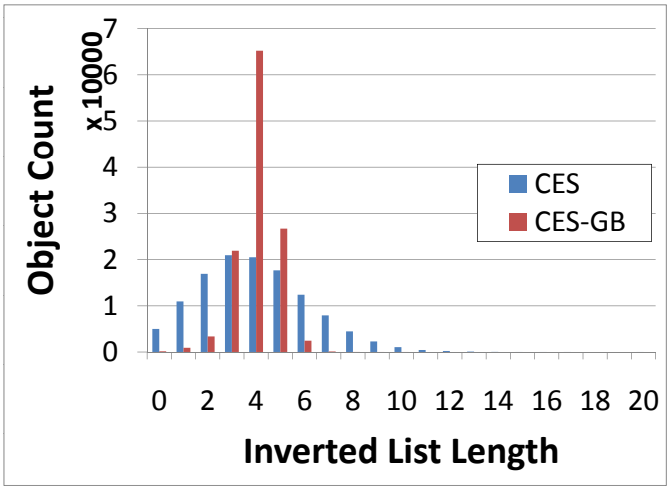

Figure 6.15 Histograms of object count versus inverted neighbor list length for the KDDCup dataset, with neighbor list size $\lambda = 20$ and a cache proportion of 20%.

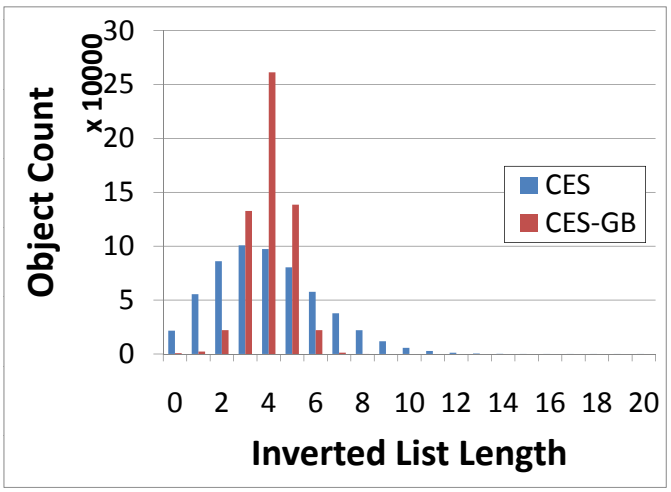

Figure 6.16 Histograms of object count versus inverted neighbor list length for the CoverType dataset, with neighbor list size $\lambda = 20$ and a cache proportion of 20%.

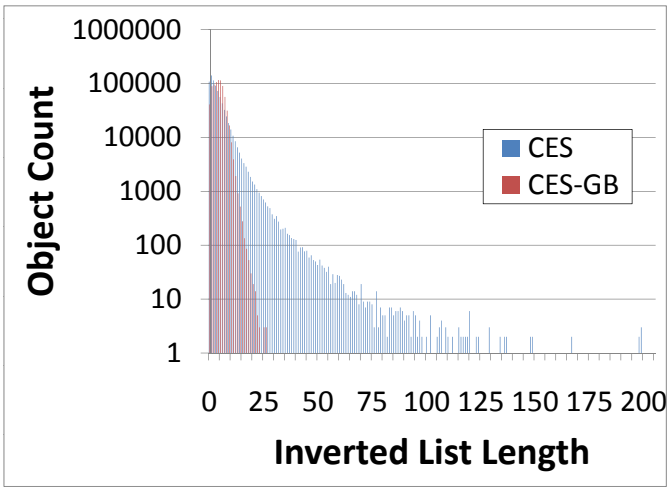

Figure 6.17 Histograms of object count versus inverted neighbor list length for the RCV1 dataset, with neighbor list size $\lambda = 20$ and a cache proportion of 20%. For the CES method, although some inverted lists had lengths in the range 200 to 600, histogram bars are shown only for lists of lengths up to 200.

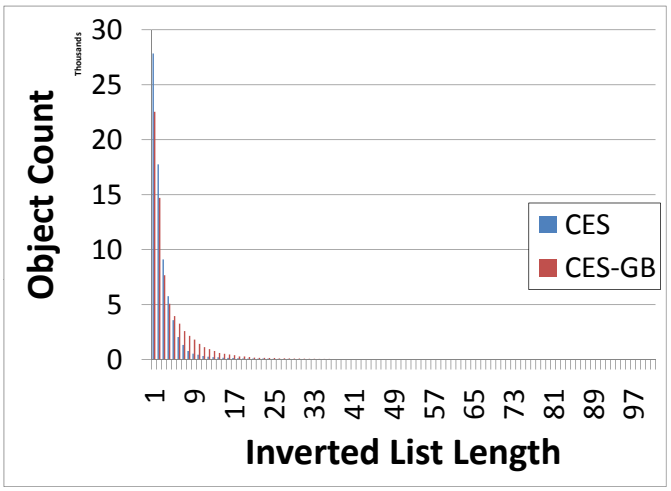

Figure 6.18 Histograms of object count versus inverted neighbor list length for the Jester dataset, with neighbor list size $\lambda = 20$ and a cache proportion of 20%.

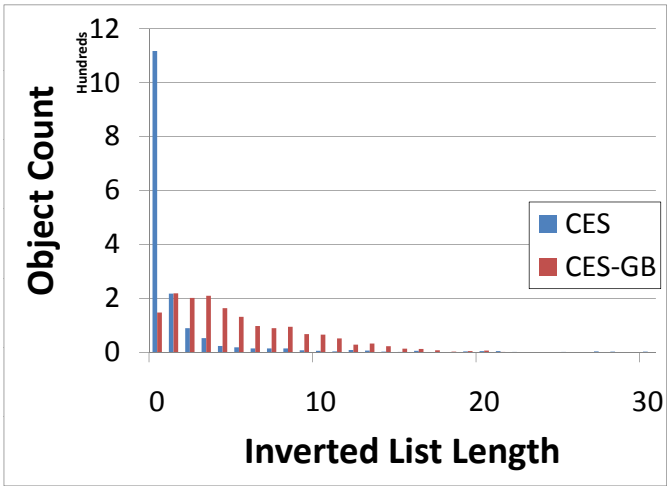

Figure 6.19 Histograms of object count versus inverted neighbor list length for the MovieLens dataset, with neighbor list size $\lambda = 20$ and a cache proportion of 20%.

Further experiments were conducted to show the relationship between the recall rates produced and the lengths of the inverted neighbor list associated with query objects. In Figures 6.20, 6.21, 6.22 and 6.23 plots are provided showing the average recall values as a function of query inverted list size. The scaling of the plot along the x-axis is expressed as a cumulative proportion of the total number of objects. For example, a plot point of $(0.7, 0.75)$ would indicate that an average recall of 75% is attained for queries based at the data objects of a common inverted neighbor length, and that objects of that length occupy up to the 70th percentile in the ordering of objects by inverted neighbor list length. The experimental results show flatter, more consistent performance for CES-GB as compared to CES. The query coverage of CES-GB is also much larger and more reliable than that of CES. In the rare cases where the inverted neighbor list of the query object is very large sizes, CES provides a better recall than CES-GB. However, the experiments clearly show that this occasional high performance by CES comes at the expense of coverage and consistency.

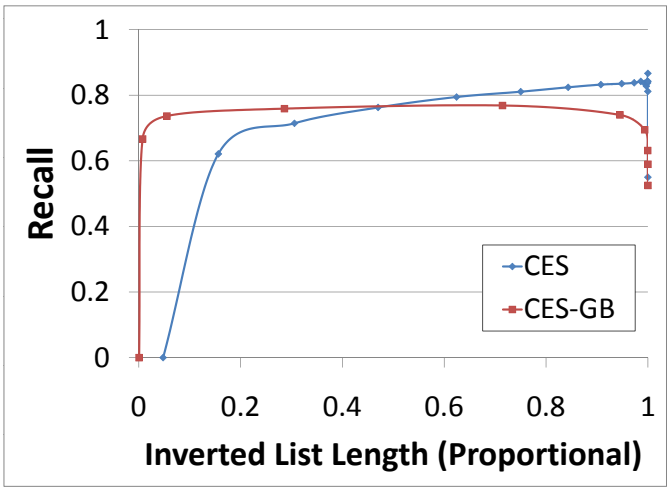

Figure 6.20 Average recall as a function of query inverted list size for the ALOI data set, with $k = \lambda = 20$ and a cache proportion of 20%.

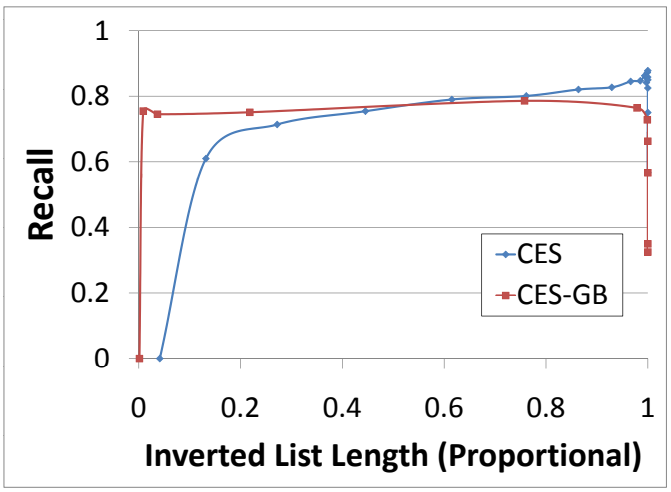

Figure 6.21 Average recall as a function of query inverted list size for the KDDCup data set, with $k = \lambda = 20$ and a cache proportion of 20%.

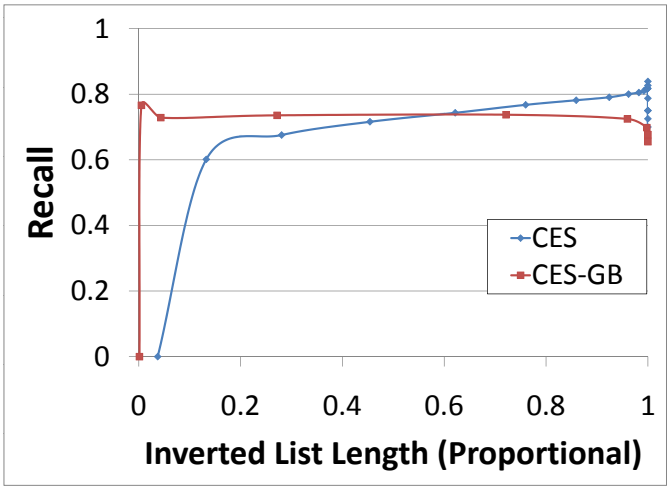

Figure 6.22 Average recall as a function of query inverted list size for the CoverType data set, with $k = \lambda = 20$ and a cache proportion of 20%.

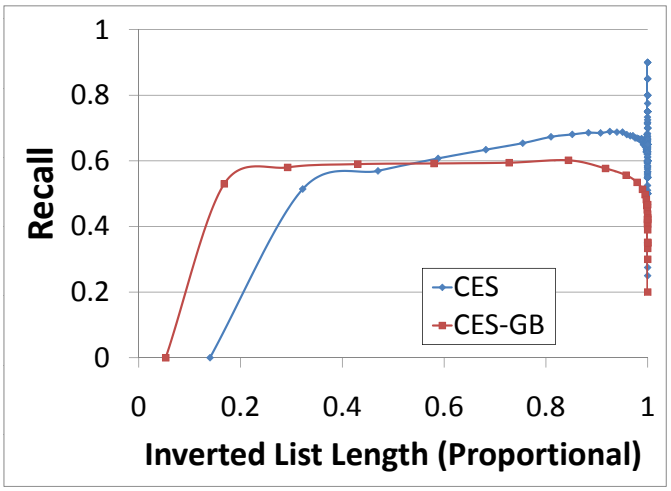

Figure 6.23 Average recall as a function of query inverted list size for the RCV1 data set, with $k = \lambda = 20$ and a cache proportion of 20%.

6.5 Summary

This chapter proposes a greedy balancing strategy, CES-GB, for the selection of appropriate cache data in order to answer the largest possible number of queries. The active strategy

presented in [49] has been shown to depend on the frequency in which the query object appears together with result objects in the lists stored in the cache. This frequency is equal to the length of the inverted lists associated with cached result lists. The quality of the result has been shown to be lower for when the query object is associated with a shorter inverted list, and better when it is associated with a longer inverted list. Furthermore, no result can be estimated when the query object is associated with an empty inverted list. The proposed greedy balancing heuristic for the selection of the cache content provides a good coverage over the range of possible queries, and improves both the hit rate and average recall even for small cache sizes.

The main contribution of the CES-GB algorithm is that it balances the size of the inverted cache lists through reduction in variance of the lengths of these lists, thereby balancing the frequency of appearance of objects in the cached top-k neighbor lists. By achieving a better inverted list balance, CES-GB provides a better uniform coverage of the query range, and increases the spatial locality from which most if not all query results can be actively generated.

CES-GB provides significant improvement in the hit rate and average recall for small caches. Since the size of cache memory is usually much smaller than the total dataset size, this approach can have a great practical impact. Even for small caches, CES-GB may be sufficient to answer all queries actively, without ever referring to the original dataset. This form of active caching therefore has the potential to serve as a scalability technique. With the explosive growth of data repositories and the popularity of similarity-based applications, the CES-GB approach opens doors for new forms of indices based on data sampling.

Algorithm 1 GreedyBalance

Input S: set of objects.

F: file containing λ-NN lists for all objects of S.

m: target cache set size.

Output $SelectedList$: list of selected cache objects.

Initialize $SelectedList \leftarrow \emptyset$.
Initialize min-heap $ScoreHeap$ with a single node with score 0.
Associate the node of $ScoreHeap$ with a structure $RankedList(0)$ holding all λ-NN lists from file F.
Set pointer RL to indicate that $RankedList(0)$ is associated with the top element of $ScoreHeap$.
Initialize set $HoldingSet \leftarrow \emptyset$.
For each item $s \in S$ **do**
  Initialize $\Gamma(s) \leftarrow 0$.
EndFor
While ($|SelectedList| < m$) **do**
  Select a new object s randomly from RL.
  For each item v in the neighbor list $Q(s, \lambda)$ **do**
    For each item w in the inverted neighbor list $Q^{-1}(v, \lambda)$ **do**
      If $w \notin HoldingSet$ **then**
        Insert $HoldingSet \leftarrow HoldingSet \cup \{w\}$.
        Delete w from $RankedList(\Gamma(w))$.
        If $RankedList(\Gamma(w)) = \emptyset$ **then**
          Delete $RankedList(\Gamma(w))$.
          Update $ScoreHeap$, and (if necessary) RL.
        EndIf
      EndIf
      Increment $\Gamma(w)$ by 1.
    EndFor
  EndFor
  For each item $w \neq s$ in $HoldingSet$ **do**
    If $RankedList(\Gamma(w)) = \emptyset$ **then**
      Create structure $RankedList(\Gamma(w)) \leftarrow \{w\}$.
      Insert $RankedList(\Gamma(w))$ into $ScoreHeap$.
    EndIf
    Insert $RankedList(\Gamma(w)) \leftarrow RankedList(\Gamma(w)) \cup \{w\}$.
  EndFor
  Insert object s into $SelectedList$.
  Reset $HoldingSet \leftarrow \emptyset$.
EndWhile
Return $SelectedList$.

CHAPTER 7

SUMMARY, LIMITATIONS AND FUTURE WORK

This chapter will summarize the major findings of this study, discuss the results in terms of theoretical and practical implications, outline the contributions of this study, and discuss the limitations and possible future research directions.

7.1 Summary

This study proposes a caching solution for the efficiency issue in recommendation systems. It aims at finding a caching solution that can work with any type of recommender system as well as any type of distance matrix i.e., metric & non-metric. Normally only a small proportion of the entire dataset is cached and traditionally this cached information is used to answer most popular queries. This study, on the other hand, is focused on finding a more scalable solution that can help to answer most, if not all, of the queries regardless of their popularity using cached information. Three major research issues in this study are: How to design an effective and efficient caching solution for recommender systems? How to design a more general and effective similarity measure for active caching? and How to select the objects in the cache for a caching with no replacement?

The first research question, how to design an effective and efficient caching solution for recommender systems, is addressed through partial order based approach proposed in the Chapter 4. The proposed solution helps to answer most of the queries in a dataset using the small subset of data available in the cache. It can not only answer queries whose result is readily available in the cache but can also actively process answers for non-cached queries and in a sense cache acts in a limited query processing role. This solution does not rely on popular or most recent data hence, works well even in the absence of any access patterns. Also the solution is not dependent on any type of recommender system

or type of distance matrix and performs well with variety of datasets. Partial order based active caching approach uses monotonicity amongst the cached information to estimate the answers for non-cached queries. This approach will result in lower recall for estimated answers in datasets having lower monotonicity. The second research question addresses the issue of how to design a more general and effective similarity measure for active caching. Shared neighbor similarity measure for active caching proposed in Chapter 5 addresses this question. A general model, the Cache-Estimated Significance (CES), is proposed for the estimation of the results of similarity queries using shared-neighbor similarity measures on cached information. The proposed method is general in that it does not require that the features be drawn from a metric space, nor does it require that the partial orders induced by the similarity measure be monotonic. It successfully improves the recall rates for queries whose results are estimated from the cache. The third and final major research question addressed by this study is how to select the objects in the cache for a caching with no replacement. This question is addressed in the Chapter 6 of this study. It proposes a greedy balancing cache selection policy which helps to provide better over all coverage of the data and increases the spatial locality in the cache. This approach uses a greedy balancing strategy, CES-GB, for the selection of appropriate cache data in order to answer the largest possible number of queries. The proposed greedy balancing heuristic for the selection of the cache content provides a good coverage over the range of possible queries, and improves both the hit rate and average recall even for small cache sizes.

Active caching is an extension of the caching model whereby estimation is used to generate an answer for queries whose results are not explicitly cached, where the estimation makes use of the results cached for related queries. By answering non-cached queries along with cached queries, active caching approach offers substantial improvement over traditional caching methodologies. Active caching approach presented in this work showed very strong overall performance and provides an intriguing answer to the question of cache management for recommender systems. The experimental results show substantial

improvement in the cache hit rate while achieving high recall rates. The proposed approach can not only answer queries that exactly match the queries in the cache but also computes answers for non-cached queries hence, the cache acts in a limited query processor role.

Active caching extends the performance of a conventional cache so that whenever the target result is not explicitly available in the cache, it makes use of the stored results from previous queries to estimate the result for the current query. Whereas the conventional approach is to fill the cache with those items most likely to be requested in future queries, experimental results show that the active caching can instead support a form of *data interpolation*, in which the cache is selected so as to provide uniform coverage of the data set from which most if not all query results are actively generated. For some applications, it may even suffice to answer all queries actively without ever referring to the original data. The proposed solution goes beyond the keyword-based cache solutions proposed for web applications — it is quite general, and independent of any method used to generate the ranked lists. In this sense, it can be applied even when non-metric and probabilistic approaches are used in generating ranked lists. Active caching could thus serve as a scalability technique, as it provides the basis of space- and time-efficient approximation of recommender system applications.

7.2 Contributions and Implications

7.2.1 Contributions

This study contributes to recommender system as well as caching domains. It provides a caching strategy that is specifically designed for recommender system queries however, it works with any application which uses top-k similarity queries (also known as k-nearest-neighbor, or k-NN queries). This caching strategy, active cache mechanism, can answer not only queries that exactly match the queries in the cache, but also act as a limited query processor by computing results for non-cached query items using information cached for other items. The main contributions of this study are:

The main contribution of the active caching approach is to provide a mechanism which assists in answering not only queries that exactly match the queries in the cache but also estimating answers for non-cached queries thus using cache in a limited query processor role. This approach does not assume any knowledge of the methods or similarity measures used, and as such can be applied even when non-metric and probabilistic approaches are used to produce query results. Whereas the conventional approach is to fill the cache with those items most likely to be requested in future queries, active caching can instead support a form of *data interpolation*, in which the cache is selected so as to provide uniform coverage of the data set from which most if not all query results are actively generated. For some applications, it may even suffice to answer all similarity queries actively, without ever referring to the original data. Active caching could thus serve as a scalability technique, as it provides the basis of space- and time-efficient approximation of large databases [105].

The main contribution of the shared neighbor approach is to facilitate the design of shared-neighbor ranking formulae for active caching that allow for variation of parameters. The ranking function can correct for bias relating to variations in such quantities as the size of the cache, the length of ranked lists stored in the cache, and the number of items requested by the query, all without any knowledge of the actual similarity values [49].

The main contribution of the greedy balancing cache selection policy is that it balances the size of the inverted cache lists through reduction in variance of the lengths of these lists, thereby balancing the frequency of appearance of objects in the cached top-k neighbor lists. By achieving a better inverted list balance, it provides a better uniform coverage of the query range, and increases the spatial locality from which most if not all query results can be actively generated. CES-GB provides significant improvement in the hit rate and average recall for small caches. Since the size of cache memory is usually much smaller than the total dataset size, this approach can have a great practical impact. Even for small caches, CES-GB may be sufficient to answer all queries actively, without ever referring to

the original dataset. This form of active caching therefore has the potential to serve as a scalability technique. With the explosive growth of data repositories and the popularity of similarity-based applications, the CES-GB approach opens doors for new forms of indices based on data sampling [45].

7.2.2 Implications

This study has several implications in various research fields. These implications are possible by applying the proposed caching solution with other query types. Furthermore, proposed methods can also be adopted in other areas of research.

Active caching approach presented in this work is primarily designed to work with recommender systems and showed very strong overall performance for recommender systems. This approach can also work with any application which uses top-k similarity queries (also known as k-nearest-neighbor, or k-NN queries). As such this approach can be easily and effectively used with similar other applications like contextual advertising, image retrieval etc. which use top-k similarity queries. Another possible implication of active caching approach is to modify the solution so that it can work with other types of queries. Modifying and using this approach with keyword queries can significantly improve the performance of applications like search engines, digital libraries etc. Another implication of this work is possible by using this approach with boolean queries which can make database caching an effective approach to achieve high scalability and performance.

The main contribution of the shared neighbor approach is to facilitate the design of shared-neighbor ranking formulae for active caching. Shared-neighbor similarity measure assess the statistical significance of the relationship between objects based on their shared neighborhood. This concept provides new directions in various domains and can help to introduce new approaches based on shared-neighbor information. A possible implication of shared-neighbor approach is the development of a new type of recommender system

which will be based on shared-neighbor information. This type of recommender system can deduce from rich sources of relationships, text, images, media etc. using shared-neighbor information and provide effective cross-genre recommendations.

Greedy balancing approach introduced in this work successfully provides a better uniform coverage of the dataset. This approach has great implications in the areas of data summarization and data sampling. Greedy balancing approach can help in computing data summarization in very large multi-dimensional datasets like data warehouses which otherwise require a very powerful and time consuming operations. GB approach also opens doors for new forms of indices based on data sampling where a better uniform coverage of the dataset can make these indices much more effective.

7.3 Limitations and Future Work

7.3.1 Limitations

Similar to any other caching solution, this approach also incurs an overhead in case of cache miss. This overhead is due to the fact that the answer is checked in the cache first and if it is not available then requested from the database. Active caching approach on the other hand has much lower overhead because the number of cache misses are very low. The active caching solution proposed in this study is mainly targeted for recommender systems but can be used with other nearest neighbor applications. Nevertheless, for any mission critical application, this solution should be used with care as the estimated answer for non-cached queries processed from the cache is not always exactly similar to the result if fetched from the database. The potential of the proposed active caching solution for the large commercial data sets deserves further experimental investigation. Another limitation of current approach is that it works with a fix set of objects. Objects are selected upfront to be populated in the cache and any new addition or deletion of objects requires that cache be populated again. Hence, any such applications where set of object constantly change,

cache should be populated periodically so that any new objects have equal chance of being selected from the cache.

7.3.2 Future Work

In the future, author is interested in applying this approach with other similar type of applications to see its practical implications. Author is also interested in investigating how this approach can be modified to work with other types of queries e.g. keyword queries and boolean queries. Performance of applications like search engines, voice recognition, face detection etc. can be significantly improved if this approach can be modified to work with these systems. Author is particularly interested in applying active caching approach with search engine queries. Furthermore, author would like to investigate a new type of recommender system based on shared-neighbor information.

BIBLIOGRAPHY

[1] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams. Removal policies in network caches for world-wide web documents. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 293–305, New York, NY, USA, 1996. ACM.

[2] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. Technical report, Blacksburg, VA, USA, 1995.

[3] N. R. Adam, V. Atluri, and I. Adiwijaya. Si in digital libraries. *Communications of the ACM*, 43(6):64–72, 2000.

[4] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, 2005.

[5] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, 1990.

[6] J. Alspector, A. Koicz, and N. Karunanithi. Feature-based and clique-based user models for movie selection: A comparative study. *User Modeling and User-Adapted Interaction*, 7:279–304, 1997.

[7] J. Alspector, A. Kolcz, and N. Karunanithi. Comparing feature-based and clique-based user models for movie selection. In *DL '98: Proceedings of the third ACM conference on Digital libraries*, pages 11–18, New York, NY, USA, 1998. ACM.

[8] J. L. Ambite, N. Ashish, G. Barish, C. A. Knoblock, S. Minton, P. J. Modi, I. Muslea, A. Philpot, and S. Tejada. Ariadne: a system for constructing mediators for internet sources. *SIGMOD Rec.*, 27(2):561–563, 1998.

[9] M. F. Arlitt, L. Cherkasova, J. Dilley, R. J. Friedrich, and T. Y. Jin. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Perform. Eval*, 27(4):3–11, 2000.

[10] M. Balabanovic and Y. Shoham. Fab: Content-based, collaborative recommendation. *Comm. ACM*, 40(3):66–72, 1997.

[11] C. Basu, H. Hirsh, and W. W. Cohen. Recommendation as classification: Using social and content-based information in recommendation. In *In Proceedings of the Fifteenth National Conference on Artificial Intelligence*, July 1998.

[12] D. Bawden, C. Holtham, and N. Courtney. Perspectives on information overload. In *Aslib Proceedings*, September 1999.

[13] N. J. Belkin and W. B. Croft. Information filtering and information retrieval: two sides of the same coin? *Commun. ACM*, 35(12):29–38, 1992.

[14] M. Bhatt. Locality of reference. *Pattern Languages of Programming*, September 1997.

[15] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 1–16, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[16] N. Boujemaa, J. Fauqueur, M. Ferecatu, F. Fleuret, V. Gouet, B. Le Saux, and H. Sahbi. Ikona: Interactive generic and specific image retrieval. In *Intern. Workshop on Multimedia Content-Based Indexing and Retrieval (MMCBIR)*, 2001.

[17] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of The 14th Conference on Uncertainty in Artificual Intelligence*, 1998.

[18] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM 99*, 1999.

[19] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International WWW Conference*, 1998.

[20] R. Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, 2002.

[21] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[22] P. Cao, J. Zhang, and K. Beach. Active cache: caching dynamic contents on the web. *Distributed Systems Engineering*, 6(1):43–50, 1999.

[23] E. J. Castellano and L. Martnez. A web-decision support system based on collaborative filtering for academic orientation. *Journal of Universal Computer Science*, 15(14), 2009.

[24] C. Chen, X. Fu, P. Liu, and B. Chen. A cooperative browser-level web caching system based on chord. pages 93 –97, apr. 2008.

[25] B. Chidlovskii and U. Borghoff. Signature file methods for semantic query caching. In *Proc. of the 2nd European Conf. on Digital Libraries*, September 1998.

[26] B. Chidlovskii and U. Borghoff. Semantic caching of web queries. *The VLDB Journal*, 9(1):2–17, 2000.

[27] B. Chidlovskii, C. Roncancio, and M. Schneider. Cache mechanism for heterogeneous web querying. In *Proc. 8th World Wide Web Conference (WWW8)*, May 1999.

[28] W. W. Chu, Q. Chen, and A. Huang. Query answeing via cooperative data inference. *JIIS*, (3):57–87, 1994.

[29] I. Chun and I. Hong. The implementation of knowledge-based recommender system for electronic com-merce using java expert system library. In *IEEE International Symposium on industrial electronics*, pages 12–16, 2001.

[30] M. Claypool, A. Gokhale, T. Miranda, P. Murnikov, D. Netes, and M. Sartin. Combining content-based and collaborative filters in an online newspaper, 1999.

[31] M. K. Condliff, D. D. Lewis, and D. Madigan. Bayesian mixed-effects models for recommender systems. In *In ACM SIGIR 99 Workshop on Recommender Systems: Algorithms and Evaluation*, 1999.

[32] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of www client based traces. Technical report, Report TR-95-010. Boston University, Computer Science Dept., 1995.

[33] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *In Proceedings of the Conference on Very Large Data Bases (VLDB)*, 1996.

[34] X. Ding, S. Jiang, and F. Chen. A buffer cache management scheme exploiting both temporal and spatial localities. *Trans. Storage*, 3(2):5, 2007.

[35] B. M. Duska, D. Marwood, and M. J. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *In Proceedings of USENIX Symposium on Internet Technology and Systems*, December 1997.

[36] T. Fagni, R. Perego, F. Silvestri, S. Orlando, U. Ca, and F. Venezia. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst*, 24:2006, 2006.

[37] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogenous environments. In *Proceedings of the IEEE infocomm99*, March 1999.

[38] K. Funakoshi and T. Ohguro. A content-based collaborative recommender system with detailed use of evaluations. In *4th Int Conf of Knoledge-Based Intelligent Engineering Systems and Allied Technologies (KES2000)*, 2000.

[39] J. Geusebroek, G. Burghouts, and A. Smeulders. The amsterdam library of object images. *Int. J. Comput. Vision*, 611:103–112, 2005.

[40] S. Glassman. A caching relay for the world wide web. In *Selected papers of the first conference on World-Wide Web*, pages 165–173, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V.

[41] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, 1992.

[42] D. Goldberg, D. Nichols, B. M. Oki, and D. B. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[43] S. D. Gribble and E. A. Brewer. System design issues for internet middleware services: deductions from a large client trace. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 19–19, Berkeley, CA, USA, 1997. USENIX Association.

[44] X. Gu, I. Christopher, T. Bai, C. Zhang, and C. Ding. A component model of spatial locality. In *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, pages 99–108, New York, NY, USA, 2009. ACM.

[45] H. Han, M. H., V. Oria, and U. Qasim. Caching without replacement for approximate similarity search. *Proceedings of the Very Large Database (In process)*, 2011.

[46] J. Herlocker, J. Konstan, A. Borchers, and J. Riedl. Framework for performing collaborative filtering. In *Proceedings of the 1999 Conference on Research and Development in Information Retrieval*, Aug 1999.

[47] W. Hill, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Proc. Conf. Human Factors in Computing Systems.*, January 1995.

[48] M. Houle. The relevant-set correlation model for data clustering. *Statistical Analysis and Data Mining*, 1(3):157–176, 2008.

[49] M. E. Houle, V. Oria, and U. Qasim. Active caching for similarity queries based on shared-neighbor information. In *CIKM 2010: The 19th ACM International Conference on Information and Knowledge Management*, 2010.

[50] M. E. Houle and J. Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *International Conference on Data Engineering*, pages 619–630, 2005.

[51] E. M. Housman and E. D. Kaskela. State of the art in selective dissemination of information. *Engineering Writing and Speech, IEEE Transactions on*, 13(2):78 –83, Sep. 1970.

[52] Z. Huang, H. Chen, and D. Zeng. Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering. *ACM Transactions on Information Systems*, 22:116–142, 2004.

[53] X. Long J. Zhang and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. 17th Int. Conf. on World Wide Web*, 2008.

[54] R. A. Jarvis and E. A. Patrick. Clustering using a similarity measure based on shared nearest neighbors. *IEEE Trans. Comput.*, C-22:1025–1973, 1973.

[55] S. Jin and A. Bestavros. Greedydual: Web caching algorithms exploiting the two sources of temporal locality in web request streams. In *In Proceedings of the 5th International Web Caching and Content Delivery Workshop.*, 2000.

[56] B. Jnsson, M. Arinbjarnar, B. rsson, M. J. Franklin, and D. Srivastava. Performance and overhead of semantic cache management. *ACM Trans. Interet Technol.*, 6(3):302–331, 2006.

[57] M. Kampe and F. Dahlgren. Exploration of the spatial locality on emerging applications and the consequences for cache performance. *Parallel and Distributed Processing Symposium, International*, 0:163, 2000.

[58] S. Kangas. Collaborative filtering and recommendation systems. Technical report, VTT Information Technology, 2002.

[59] G. Karakostas and D. N. Serpanos. Exploitation of different types of locality for web caches. In *Proceedings of the Seventh International Symposium on Computers and Communications*, 2002.

[60] P. Kazienko and M. Kiewra. *Personalized Recommendation of Web Pages*, chapter Chapter 10 in Intelligent Technologies for Inconsistent Knowledge Processing. Advanced Knowledge International, Adelaide, South Australia, 2004.

[61] M. G. Kendall and J. D. Gibbons. *Rank Correlation Methods*. Charles Griffin, 5th edition, 1990.

[62] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 19–34, New York, NY, USA, 1996. ACM.

[63] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. Grouplens: Applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87, 1997.

[64] B. Krishnamurthy and J. Rexford. *Web protocols and practice: HTTP/1.1, Networking protocols, caching, and traffic measurement*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[65] T. M. Kroeger and D. D. E. Long. Predicting file system actions from prior events. In *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 26–26, Berkeley, CA, USA, 1996. USENIX Association.

[66] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[67] B. Krulwich and C. Burkey. The infofinder agent: learning user interests through heuristic phrase extraction. *IEEE Expert*, 12(5):22 –27, Sep. 1997.

[68] A. Kumar, P. Jalote, and D. Gupta. A server side caching scheme for corba. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 58–65, Washington, DC, USA, 2004. IEEE Computer Society.

[69] M. Steinbach L. Ertöz and V. Kumar. Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. In *Proc. of 3rd SIAM Intern. Conf. on Data Mining*, 2003.

[70] K. Lang. Newsweeder: Learning to filter netnews. In *in Proceedings of the 12th International Machine Learning Conference (ML95*, 1995.

[71] K. C. K. Lee, H. V. Leong, and A. Si. Semantic query caching in a mobile environment. *Mobile Computing and Comm.*, 3(2):28–36, 1999.

[72] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of USENIX 1997 Annual Technical Conference, USENIX*, January 1997.

[73] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias. Design and performance of web server accelerator. In *Proceedings of Infocom'99*, 1999.

[74] D. Lewis, Y. Yang, T. Rose, and F. Li. A new benchmark collection for text categoriza-tion research. *Journal of Machine Learning Research*, 5:361–397, 2004.

[75] Z. Li and I. Im. Recommender systems: A framework and research issues. In *Americas Conference on Information Systems (AMCIS)*, 2002.

[76] G. D. Linden, J. A. Jacobi, and E. A. Benson. Collaborative recommendations using item-to-item similarity mappings. US Patent 6,266,649 (to Amazon.com), Patent and Trademark Office, Washington, D.C, 2001.

[77] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international conference on World Wide Web*, May 2005.

[78] T. Loukopoulos, P. Kalnis, I. Ahmad, and D. Papadias. Active caching of on-line-analytical-processing queries in www proxies. In *ICPP '01: Proceedings of the International Conference on Parallel Processing*, page 419, Washington, DC, USA, 2001. IEEE Computer Society.

[79] C. Luk and T. C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2), 1999.

[80] Q. Luo. Proxy caching for database-backed web sites. In Xueyan Tang, Jianliang Xu, and Samuel T. Chanson, editors, *Web Content Delivery*, volume 2 of *Web Information Systems Engineering and Internet Technologies Book Series*, pages 153–174. Springer US, 2005.

[81] Q. Luo and J. F. Naughton. Form-based proxy caching for database-backed web sites. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 191–200, September 2001.

[82] Q. Luo, J. F. Naughton, R. Krishnamurthy, P. Cao, and Y. Li. Active query caching for database web servers. In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 92–104, London, UK, 2001. Springer-Verlag.

[83] Q. Luo, J. F. Naughton, and W. Xue. Form-based proxy caching for database-backed web sites: keywords and functions. *The VLDB Journal*, 17(3):489–513, 2008.

[84] J. Sander M. Ester, H. P. Kriegel and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of 2nd Int. Conf. on Knowl. Discovery and Data Mining*, 1996.

[85] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance issues of enterprise level web proxies. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 13–23, New York, NY, USA, 1997. ACM.

[86] E. Markatos. On caching search engine query results. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, May 2000.

[87] E. Markatos and C. Chronaki. A top-10 approach to prefetching the web. In *Proceedings of the 8th Annual Conference of the Internet Society (INET '98)*, 1998.

[88] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[89] P. Melville, R. J. Mooney, and R. Nagarajan. Content-boosted collaborative filtering for improved recommendations. In *in Eighteenth National Conference on Artificial Intelligence*, pages 187–192, 2002.

[90] D. A. Menasc and V. Akula. Improving the performance of online auctions through server-side activity-based caching, world wide web journal.

[91] R.V. Meteren and M.V. Someren. Using content-based filtering for recommendation. In *Machine Learning in the New Information Age MLnet / ECML2000 Workshop*, 2000.

[92] D. B. Michael. Learning collaborative information filters, 1998.

[93] B. N. Miller. Toward a personal recommender system. In *PhD Thesis*. University of Minnesota, 2003.

[94] M. Montaner, B. López, and J. L. De La Rosa. A taxonomy of recommender agents on the internet. *Artif. Intell. Rev.*, 19(4):285–330, 2003.

[95] R. J. Mooney and L. Roy. Content-based book recommending using learning for text categorization. In *DL '00: Proceedings of the fifth ACM conference on Digital libraries*, pages 195–204, New York, NY, USA, 2000. ACM.

[96] D. Moore. *Basic Practice of Statistics*. W.H. Freeman Company, 4th edition, 2006.

[97] K. Musial. Recommendation system for online social network. Master's thesis, School of Engineering Blekinge Institute of Technology Box 520, SE, 2006.

[98] S. Nagaraj. *Web Caching and Its Applications*. Kluwer, Norwell, 2004.

[99] N. Osawa, T. Yuba, and K. Hakozaki. Generational replacement schemes for a www caching proxy server. In *HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 940–949, London, UK, 1997. Springer-Verlag.

[100] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, 1996.

[101] M. Pazzani, J. Muramatsu, and D. Billsus. Syskill webert: Identifying interesting web sites. In *In Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 54–61. AAAI Press, 1996.

[102] S. Perugini, M. A. Gonçalves, and E. A. Fox. Recommender systems research: A connection-centric survey. *J. Intell. Inf. Syst.*, 23(2):107–143, 2004.

[103] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.

[104] E. W. Pugh, L. R. Johnson, and J. H. Palmer. Ibm's 360 and early 370 systems (history of computing). Technical report, The MIT Press, 1991.

[105] U. Qasim, V. Oria, Y. B. Wu, M. E. Houle, and M. T. Özsu. A partial-order based active cache for recommender systems. In *RecSys '09: Proceedings of the third ACM conference on Recommender systems*, pages 209–212, New York, NY, USA, 2009. ACM.

[106] Q. Ren, M. H. Dunham, and V. Kumar. Semantic caching and query processing. *IEEE Trans. on Knowl. and Data Eng.*, 15(1):192–210, 2003.

[107] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, New York, NY, USA, 1994. ACM.

[108] P. Resnick and H. R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, 1997.

[109] E. Rich. Users are individuals: Individualizing user models. *Journal of Man-Machine Studies*, 18:199–214, 1983.

[110] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. Technical report, Technical Report. University College London, Department of Computer Science. UK, 1998.

[111] A. Rousskov and V. Soloviev. On performance of caching proxies. In *Proceedings of the ACM SIGMETRICS Conference*, June 1998.

[112] R. Rastogi S. Guha and K. Shim. Rock: a robust clustering algorithm for categorical attributes. *Inform. Sys.*, 25:345–366, 2000.

[113] M. E. Saleh, A. A. Nabi, and A. B. Mohamed. A design and implementation model for web caching using server url rewriting world academy of science, 2009.

[114] G. Salton. *The SMART Retrieval System — Experiments in Automatic Document Processing*. Prentice-Hall, 1971.

[115] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[116] P. Saraiva, E. Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Ribeiro-Neto. Rank-preserving two-level caching for scalable search engines. In *In Proc. 24rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 51–58, 2001.

[117] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. T. Riedl. Application of dimensionality reduction in recommender system – a case study. In *IN ACM WEBKDD WORKSHOP*, 2000.

[118] B. M. Sarwar, J. A. Konstan, A. Borchers, J. Herlocker, B. Miller, and J. Riedl. Using filtering agents to improve prediction quality in the grouplens research collaborative filtering system. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 345–354, New York, NY, USA, 1998. ACM.

[119] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen. Collaborative filtering recommender systems. pages 291–324, 2007.

[120] J. B. Schafer, J. Konstan, and J. Riedi. Recommender systems in e-commerce. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166, New York, NY, USA, 1999. ACM.

[121] J. B. Schafer, J. A. Konstan, and J. Riedl. E-commerce recommendation applications. *Data Mining and Knowledge Discovery*, 5(1):115–153, 2001.

[122] K. Sequeira, M. Zaki, B. Szymanski, and C. Carothers. Improving spatial locality of programs via data mining. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 649–654, New York, NY, USA, 2003. ACM.

[123] D. N. Serpanos and W. Wolf. Caching web objects using zipfs law. In *In Proceedings of SPIE, Vol. 3527, Technical Conference 3527: Multimedia Storage and Archiving Systems III*, November 1998.

[124] U. Shardanand and P. Maes. Social information filtering: algorithms for automating "word of mouth". In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 210–217, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[125] B. Shen. Meta-caching and meta-transcoding for server-side service proxy. volume 1, pages I – 457–60 vol.1, jul. 2003.

[126] E. Shriver, C. Small, and K. Smith. Why does file system prefetching work. In *In Proceedings of the 1999 USENIX Annual Technical Conference*, 1999.

[127] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large altavista query log. Technical report, Technical Report SRC Technical Note 1998-014, 1998.

[128] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[129] I. Soboroff, C. Nicholas, and C. K. Nicholas. Combining content and collaboration in text filtering. In *In Proceedings of the IJCAI99 Workshop on Machine Learning for Information Filtering*, pages 86–91, 1999.

[130] I. Tatarinov, A. Rousskov, and V. Soloviev. Static caching in web servers, 1997.

[131] W. Teng and C. Chang. Integrating web caching and web prefetching in client-side proxies. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):444–455, 2005. Fellow-Chen, Ming-Syan.

[132] R. Torres, S. M. McNee, M. Abel, J. A. Konstan, and J. Riedl. Enhancing digital libraries with techlens+. In *JCDL '04: Proceedings of the 4th ACM/IEEE-CS joint conference on Digital libraries*, pages 228–236, New York, NY, USA, 2004. ACM.

[133] B. Towle and C. Quinn. Knowledge-based recommender systems using explict user models. In *AAAI Workshop on Knolwedge-Based Electronic Markets*, 2000.

[134] A. Vakali. Lru-based algorithms for web cache replacement. In *EC-WEB '00: Proceedings of the First International Conference on Electronic Commerce and Web Technologies*, pages 409–418, London, UK, 2000. Springer-Verlag.

[135] S. Vucetic and Z. Obradovic. A regression-based approach for scaling-up personalized recommender systems in e-commerce. In *ACM WebKDD 2000 Web Mining for E-Commerce Workshop*, 2000.

[136] T. C. Will, A. Srinivasan, M. Bieber, I. Im, V. Oria, and Y. Wu. Gre: hybrid recommendations for nsdl collections. In *JCDL*, pages 457–458, 2009.

[137] J. Williams. Hot technologies with a purpose. *Library Journal*, 127(2):50, Feb 2002.

[138] A. Woodruff, R. Gossweiler, J. Pitkow, E. H. Chi, and S. K. Card. Enhancing a digital book with a reading recommender. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 153–160, New York, NY, USA, 2000. ACM.

[139] Y. Xie and D. O'Hallaron. Locality for search engine queries and its implications for caching. In *Proc. INFOCOM*, 2002.

[140] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *In IEEE Infocom 2002*, pages 1238–1247, 2002.

[141] G. Xue, C. Lin, Q. Yang, W. Xi, H. Zeng, Y. Yu, and Z. Chen. Scalable collaborative filtering using cluster-based smoothing. In *In Proc. of SIGIR*, pages 114–121, 2005.

[142] T. Yoneya and H. Mamitsuka. Pure: A pubmed article recommendation system based on content-based filtering. In *Proceedings of The Seventh Annual International Workshop on Bioinformatics and Systems Biology*, 2007.

[143] Z. Zeng and B. Veeravalli. Hk/t: A novel server-side web caching strategy for multimedia applications. pages 1782 –1786, may. 2008.

[144] Y. Zhang, D. Li, and Z. Zhu. A server side caching system for efficient web map services. pages 32–37, jul. 2008.

[145] B. Zheng, W. Lee, and D. Lee. On semantic caching and query scheduling for mobile nearest-neighbor search. *Wirel. Netw.*, 10(6):653–664, 2004.