KNOWLEDGE REFRESHING: MODEL, HEURISTICS AND APPLICATIONS

by

Xiao Fang

A Dissertation Submitted to the Faculty of the

COMMITTEE ON BUSINESS ADMINISTRATION

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY
WITH A MAJOR IN MANAGEMENT

In the Graduate College

THE UNIVERSITY OF ARIZONA

2003

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Xiao Fang

entitled Knowledge Refreshing: Model, Heuristics and Applications

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Olivia R. Liu Sheng, Ph.D.	7-28-03 Date
Hsinchun Chen, Ph.D.	8-7-03 Date
Sherry M.B. Thatcher, Ph.D.	7/28/03 Date
	Date
	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

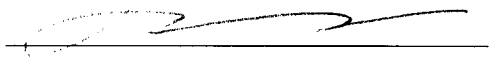
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Hsinchun Chen, Ph.D. Dissertation Director	8-7-03 Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: 

ACKNOWLEDGEMENTS

First of all, I would like to thank my dissertation advisor and mentor, Professor Olivia R. Liu Sheng, for her guidance and encouragement throughout my five years at the University of Arizona. I also thank my major committee members, Dr. Hsinchun Chen and Dr. Sherry M.B. Thatcher, and my minor committee members in the Department of Computer Science, Dr. Richard T. Snodgrass and Dr. Peter J. Downey, for their guidance and encouragement. I am fortunate to meet an excellent group of doctoral students in the department, who share their insights and ideas with me. I would like to especially thank Wei Gao, Lin Lin and Michael Chau, for the good times we spent together on and off campus.

TABLE OF CONTENTS

TABLE OF CONTENTS - *Continued*

LIST OF FIGURES

LIST OF FIGURES - *Continued*

LIST OF TABLES

LIST OF TABLES - *Continued*

ABSTRACT

With the wide application of information technology in organizations, especially the rapid growth of E-Business, masses of data have been accumulated. Knowledge Discovery in Databases (KDD) gives organizations the tools to sift through vast data stores to extract knowledge supporting organizational decision making. Most of the KDD research has assumed that data is static and focused on either efficiency improvement of the KDD process (e.g., designing more efficient KDD algorithms) or business applications of KDD. However, data is dynamic in reality (i.e., new data continuously added in). Knowledge discovered using KDD becomes obsolete rapidly, as the discovered knowledge only reflects the status of its dynamic data source when running KDD. Newly added data could bring in new knowledge or invalidate some discovered knowledge. To support effective decision making, knowledge discovered using KDD needs to be updated along with its dynamic data source. In this dissertation, we research on knowledge refreshing, which we define as the process to keep knowledge discovered using KDD up-to-date with its dynamic data source. We propose an analytical model based on the theory of Markov decision process, solutions and heuristics for the knowledge refreshing problem. We also research on how to apply KDD to such application areas as intelligent web portal design and network content management. The knowledge refreshing research identifies and solves a fundamental and general problem appearing in all KDD applications; while the applied KDD research provides a test environment for solutions resulted from the knowledge refreshing research.

CHAPTER 1: INTRODUCTION

With the wide application of information technology in organizations, especially the rapid growth of E-Business, masses of data have been accumulated. Knowledge Discovery in Databases (KDD) (Fayyad et al. 1996) gives organizations the tools to sift through vast data stores to extract knowledge supporting organizational decision making. Previous research has applied KDD in such business areas as finance (Tam and Kiang 1992, Sarkar and Sriram 2001), marketing (Cooper and Giuffrida 2000) and credit-risk evaluation (Baesens et al. 2003). Most of the KDD research has assumed that data is static and focused on either efficiency improvement of the KDD process (e.g., designing more efficient KDD algorithms) or business applications of KDD. However, data is dynamic in reality (i.e., new data continuously added in). Knowledge discovered using KDD becomes obsolete rapidly, as the discovered knowledge only reflects the status of its dynamic data source when running KDD. Newly added data could bring in new knowledge and invalidate some discovered knowledge. To support effective and efficient decision making, knowledge discovered using KDD needs to be updated along with its dynamic data source (Cooper and Giuffrida 2000). For example, as pointed out in (Bourgeois and Eisenhardt 1988, Eisenhardt 1989), one of the reasons for making poor and slow strategic decisions in high velocity industries is the usage of obsolete information/knowledge when making decisions. High velocity industries, such as the microcomputer industry, are the industries in which changes in demand, competition and technology are so rapid that information/knowledge is often inaccurate and obsolete (Bourgeois and Eisenhardt 1988). In this dissertation, we research on knowledge

refreshing, which we define as the process to keep knowledge discovered using KDD up-to-date with its dynamic data source.

1.1 Related Research On Knowledge Refreshing

As shown in Figure 1.1, the KDD process consists of such steps as data pre-processing (i.e., data selection, cleaning and transformation), data mining (i.e., extracting patterns such as decision trees from preprocessed data sources) and pattern post-processing (i.e., evaluating and interpreting patterns to generate knowledge, such as classification rules extracted from a decision tree, that can support decision making) (Fayyad et al. 1996).

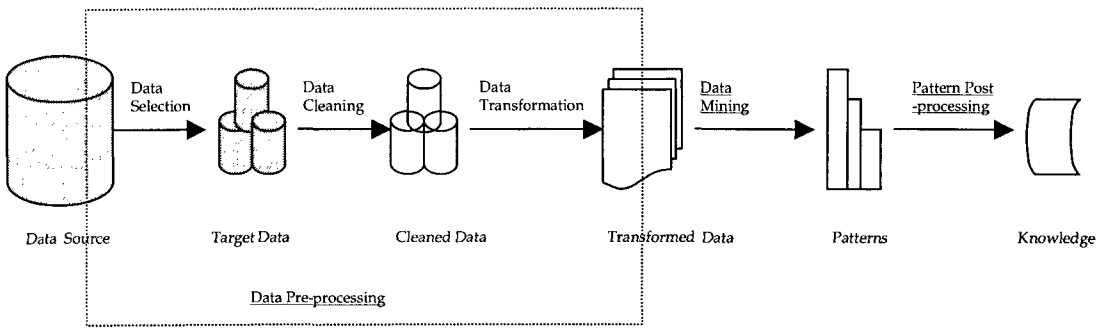


Figure 1.1: The Steps in the KDD Process

Related research, which investigated how to maintain patterns learned from data mining over a dynamic data source, can be grouped into two categories: incremental data mining and data stream mining. Incremental data mining maintains patterns over a dynamic data source by revising patterns learned from a previous run of data mining, instead of

learning from scratch. Several incremental data mining algorithms were proposed for major data mining models. For classification (Quinlan 1986), ID4 (Schlimmer and Fisher 1986) and ID5 (Utgoff 1988,1989) were developed to revise a decision tree induced from old data as new data were added in. ID4 and ID5 are based on the same idea that one can maintain instance counts for each attribute that could be a test attribute of a decision tree. They have the following common steps when handling an incoming record:

(1) Update instance counts for each potential test attribute and instance counts for observed values of each potential test attribute;

(2) If the entropy reduction of a non-test attribute is greater than that of the current test attribute, then reshape the decision tree.

ID5 differs from ID4 in its method of reshaping the decision tree. ID4 discards the subtree below the current test attribute, while ID5 pulls the best non-test attribute up to replace the current test attribute.

For association rule mining (Agrawal et al. 1993), FUP (Cheung et al. 1996) was firstly proposed to maintain large itemsets discovered using association rule mining over a dynamic data source. FUP was an iterative algorithm. In each iteration of FUP, losers which are itemsets that were large in old data source but became small after new data added in were removed, and winners which are itemsets that were small in old data source but became large after new data added in were added. A more efficient algorithm to maintain large itemsets over a dynamic data source was introduced in (Thomas et al.

1997). Based on the idea proposed in (Toivonen 1996), the algorithm scanned the whole database if and only if some itemsets in the negative border of the old large itemsets became large after new data added in. It required at most O(1) scan over the database instead of O(n) scan over the database in FUP.

In (Can 1993), an incremental clustering algorithm was proposed to maintain document clusters for an evolving document database. The algorithm first determined the number of clusters and the set of new cluster seeds after new documents were added into the database. Documents in the old clusters whose seeds are no longer seeds in the updated document database were reassigned to new clusters whose seeds covered them most. New documents were also assigned to new clusters whose seeds covered them most.

Current data-intensive applications are based on huge size data sources with rapid and continuous loading of sheer volumes of new data, termed as data streams (Babcock et al. 2002). Examples of data streams include web logs, transaction databases in retail chains, network traffic data and financial tickers. Further, for many mission-critical applications such as fraud detections in Telecom networks, it is important to be able to infer patterns from data streams online (Rastogi 2003). The characteristics of data streams, the requirements of mission-critical stream applications and architecture of modern computers have imposed the following constraints on algorithms for mining data streams:

(1) The time for processing each record in data streams must be small;

(2) The amount of memory available is limited;

(3) Single-pass algorithms: each record in a data stream can be processed only once. Once a record in a data stream is processed, it will be archived or discarded. Hence, it cannot be easily retrieved because of I/O cost.

Because of the processing speed and memory constraints, incremental data mining algorithms are not capable of maintaining real-time patterns over data streams (Domingos and Hulten 2000). Recently, a number of data stream mining algorithms were proposed to maintain real-time patterns over data steams. Data stream mining algorithms usually employed data reduction techniques (Barbara et al. 1997), such as sampling, to maintain approximate patterns over data steams in real time. Hoeffding tree algorithm was introduced in (Domingos and Hulten 2000, Hulten et al. 2001) to maintain approximate decision trees over data streams in real time. The algorithm applied the Hoeffding bound (Hoeffding 1963) to determine the number of samples required to learn a test attribute at a node. Given a data stream, the algorithm used the first ones to choose the root test attribute; and the succeeding data were passed down to the corresponding leaves and used to choose test attributes there. The space complexity of Hoeffding tree algorithm is $O(ldvc)$, where l is the number of leaves in a tree, d is the number of attributes, v is the maximum number of values per attribute and c is the number of classes.

In (Hidber 1999), an algorithm called Carma was proposed for online discovery of large itemsets. The algorithm required two scans over a data source to discover all large itemsets and their supports. Users were free to change support threshold any time during the first scan and the algorithm maintained a superset of all large itemsets. During the second scan, the algorithm returned all large itemsets and their supports according to the last user specified threshold. Obviously, the algorithm is not suitable for data stream mining, as data stream mining allows only one pass over a data source (Rastogi 2003). Another online association rule mining algorithm was proposed in (Aggarwal and Yu 2001). The algorithm could discover large itemsets and association rules in real time by pre-storing a superset of large itemsets in memory. It is not suitable for data stream mining either because of the limitation of memory. A data steam mining algorithm for association rule mining was introduced in (Manku and Motwani 2002), namely lossy counting algorithm. The algorithm employed the data structure D, which was a set of entries of the form (set, f, Δ), where *set* denoted an itemset, f denoted its frequency and Δ was the maximum possible error in f. Given a error bound ε, the algorithm only kept the itemsets with frequency f larger than or equal to εN in D, where N was the current number of records in a data stream and D was kept in main memory. For a given minimum support s, the algorithm outputted itemsets in D with frequency larger than or equal to $(s-\varepsilon)N$ as large itemsets. The answers produced by the algorithm have the following guarantees:

(1) All itemsets whose true frequency exceed sN are output;

(2) No itemset whose true frequency is less than $(s-\varepsilon)N$ is output;

(3) Estimated frequencies are less than the true frequencies by at most εN.

One of the early single-pass clustering algorithms is BIRTCH (Zhang et al. 1996). The algorithm maintained a summary of a data source, namely CF tree, in memory and then clustered the in-memory summary. Motivated by BIRTCH, a series of algorithms were introduced in (Guha et al. 2000, 2003, O'Callaghan et al. 2002) for clustering data streams. These algorithms assumed that data arrived in chunks, each of which fitting in main memory. They first clustered each chunk in main memory; then purged main memory and retained only cluster centers and their weights. Lastly, these algorithms clustered the cluster centers in main memory as the clustering result of a data stream. The algorithm can give a constant-factor approximation to the K-Median problem.

In summary, two categories of data mining algorithms for maintaining patterns over a dynamic data source were proposed in past literature: incremental data mining algorithms and data stream mining algorithms. The former returned exact patterns but required more memory and run slower, while the latter was more speed and memory efficient and suitable for mining data streams. The tradeoff was that the latter could only return approximate patterns.

1.2 Research Questions and Dissertation Structure

Both incremental data mining and data steam mining research only addressed one step in the KDD process – the data mining step, and focused on maintaining patterns over a dynamic data source. However, it is knowledge not patterns that can support organizational decision making effectively. For example, association rules (i.e., patterns) learned using association rule mining are usually too many to be handled by decision makers (Brin et al. 1997). Moreover, a substantial number of learned association rules are common sense or known to decision makers, hence, do not contribute to their decision making (Brin et al. 1997). To support effective decision making, the KDD process needs to be completed. For the above example, the pattern post-processing step in KDD needs to be executed to extract interesting association rules (i.e., knowledge) from all learned association rules, using criteria and techniques introduced in (Sliberschatz and Tuzhilin 1996, Kleinberg et al. 1998).

The KDD process cannot be fully automated, except for the data mining step. It requires people (e.g., domain experts) to help cleaning data before mining it and extracting knowledge from patterns discovered using data mining. Hence, the KDD process is a highly costly process (Manku and Motwani 2002), as personnel costs dominate equipment and computational costs (Gibson and Meter 2000). As a result, it is impractical to run KDD whenever there is an update in a data source, although it is possible to execute the data mining step whenever there is an update in a data source using data stream mining approaches. It is also unnecessary to run KDD whenever there

is an update in a data source. Such a practice often results in repetitive knowledge identical with previous KDD run because successive snapshots of real world data sources overlap considerably (Ganti et al. 1999). On the other hand, running KDD too seldom could result in losing critical knowledge, which, in turn, will impact decision making negatively. Therefore, it is critical to determine when to run KDD to optimize the trade-off between the cost of knowledge loss and the cost of running KDD.

Different from past research in the knowledge refreshing area, which focused on how to refresh knowledge, this dissertation studies when to refresh knowledge. In addition, the dissertation also identifies two important research problems in the web portal design area and the web caching area respectively and applies KDD to solve the problems. Besides our contributions to the two specific problem areas, the two KDD applications also provide a real world environment to study the knowledge refreshing problem. Specially, the dissertation addresses the following research questions:

(1) How to develop an analytical model to capture the trade-off between the cost of knowledge loss and the cost of running KDD over a time horizon? How to derive optimal knowledge refreshing policies from the analytical model?

(2) How to apply KDD to design an efficient web portal?

(3) How to apply KDD to develop an efficient web caching approach?

(4) Based on the above two KDD applications, what are the knowledge refreshing requirements for real world KDD applications? Can the analytical model capture the real world knowledge refreshing requirements? If so, how to implement the optimal knowledge refreshing policies derived in (1) to address the real world knowledge refreshing requirements?

The rest of the dissertation is organized to address the above research questions in order (Figure 1.2). To address research question (1), in Chapter 2, we develop a Markov decision process model of knowledge refreshing, derive optimal knowledge refreshing policies from the model and investigate analytic properties and performance of optimal knowledge refreshing policies. We study two real world KDD applications to address research questions (2) and (3) respectively. The two KDD applications are web mining based web portal design described in Chapter 3 and a data mining based caching approach illustrated in Chapter4. To address research question (4), in Chapter 5, we summarize the knowledge refreshing requirements in the two KDD applications and develop implementation strategy of optimal knowledge refreshing policies and heuristics based on insights generated in Chapter 2 to address these requirements. We conclude the dissertation and outline future research directions in Chapter 6. The dissertation research employs both technical research methodology such as algorithm design and system development and quantitative research methodology such as mathematical modeling, Markov decision process and simulation.

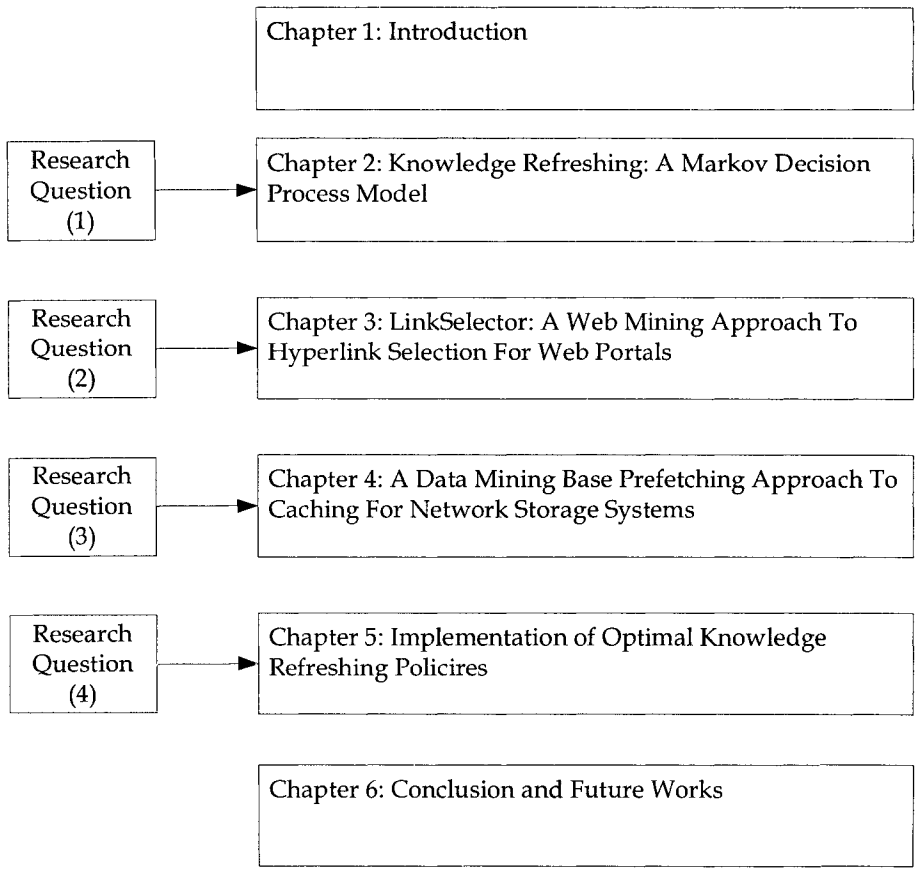


Figure 1.2: The Structure of the Dissertation

CHAPTER 2: KNOWLEDGE REFRESHING: A MARKOV DECISION PROCESS MODEL

2.1 A Markov Decision Process Model of Knowledge Refreshing

Before presenting a mathematical model of the knowledge refreshing problem addressed in this dissertation, we first give a general description of the problem. As shown in Figure 2.1, a data source evolves over time as new data continuously populated into it. To keep its knowledge base up-to-date with the data source, a KDD run has to be executed. There are queries submitted to the knowledge base to get knowledge supporting organizational decision making. At the time of a query arrival, executing KDD incurs a cost of running KDD while not executing KDD results in a cost of knowledge loss. The knowledge refreshing problem addressed in this dissertation is to find rules that can help determine time points over a time horizon to execute KDD so that the total cost, including costs of running KDD and costs of knowledge loss, over the time horizon is minimized.

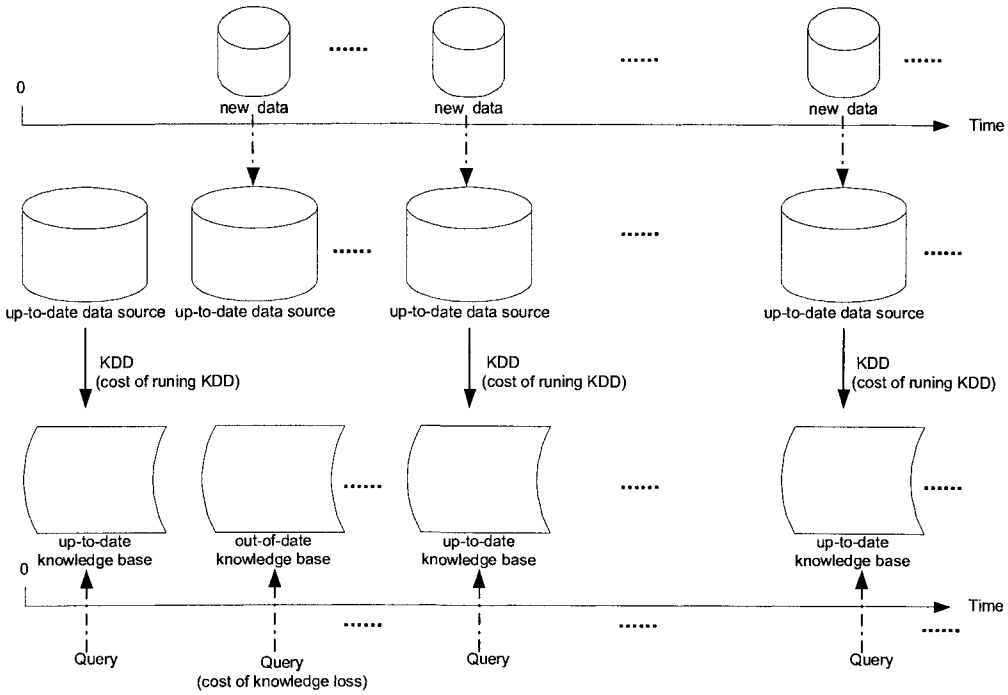


Figure 2.1: Knowledge Refreshing

The knowledge refreshing problem addressed in this dissertation can be categorized as a problem of sequential decision making under uncertainty, which is usually modeled using Markov decision process (Sutton and Barto 1998; Ross 2000). Past research has applied Markov decision process model to study optimal file migration in distributed computer systems (Liu Sheng 1992), optimal policies for database reorganization (Park et al. 1990) and adaptive inventory control (Treharne and Sox 2002) etc.. Markov decision process model usually consists of the following components: decision point, action space, state space, transition probability, cost structure and objective function. We will describe the components of the Markov decision process model for knowledge refreshing one by one below.

Two assumptions are made to make the model analytically tractable. Robustness of the model with respect to the assumptions is presented in Section 2.3. We assume that new data arrival follows a Poisson process with intensity λ_d. Decision makers query the knowledge base, which contains knowledge discovered from the data source by running KDD, to search for knowledge supporting their decision making. Queries for the knowledge base are classified into distinct types based on the following considerations. Different types of queries have different arrival rates and the same amount of knowledge loss has different impact on different types of queries. For example, queries supporting strategic decision making could have lower arrival rate than queries supporting operational decision making; while the same amount of knowledge loss could have higher impact on the former than the latter. We assume that there exists n distinct types of queries, $\{q_i\}, i = 1,2,\cdots n$, where $n \geq 1$, and each type of queries follows a Poisson process with intensity λ_{q_i} independently. Query arrivals are also independent of new data arrival. For the convenience of readers, important notations used in this chapter are summarized in Table 2.1.

Table 2.1: Notation Summary

Notation	Description
λ_d	the arrival rate of new data
$\{q_i\}, i = 1,2,\cdots n$	the set of different types of queries to the knowledge base
λ_{q_i}	the arrival rate of query type $q_i, i = 1,2,\cdots n$
A	the action space
s_m	the system state at decision point m
q_m^a	the type of the query arrived at decision point m, $q_m^a \in \{q_i\}$

l	the expected amount of knowledge loss brought in by a unit of new data
l_m	the amount of knowledge loss at decision point m
r_m	the latest decision point with action 1 (i.e., running KDD) before decision point m
d_m	the number of units of new data arrived between decision points r_m and m
S_m	the state space of s_m, $s_m \in S_m$
a_m	the action chosen at decision point m, $a_m \in A$
$P^{a_m}_{s_m s_{m+1}}$	one step transition probability from state s_m to s_{m+1} under action a_m
$c(s_m, a_m)$	the system cost incurred at decision point m
c^l_m	the cost of knowledge loss at decision point m
c_{q_i}	the cost of per unit knowledge loss for query type $q_i, i = 1,2,\cdots n$
C_k	the cost of running KDD
M	the expected number of decision points over a relative long time horizon
δ_m	the decision rule at decision point m such that $a_m = \delta_m(s_m)$
π	a refreshing policy, $\pi = (\delta_1, \delta_2, \cdots, \delta_m, \cdots, \delta_M)$
EC_π	the expected total system cost over the time horizon under refreshing policy π
Π	the set of all feasible refreshing policies
π^*	the global optimal knowledge refreshing policy, where $EC_{\pi^*} = \min_{\pi \in \Pi} EC_\pi$
$v_{s_m}(m)$	the optimal expected total system cost from decision point m to the end of the time horizon with current system state s_m
Π_f	the set of knowledge refreshing policies that refresh knowledge every fixed number of queries/decision points
π^*_f	the fixed optimal knowledge refreshing policy, $\pi^*_f \in \Pi_f$

The decision points in the model are the moments when a query for the knowledge base arrives. The system time of the model is labeled as $t = 0,1,2,\cdots,m,\cdots$, with 0 being the

starting time of the time horizon and m, where $m \geq 1$, denoting the time of decision point m or the time when the arrival of the mth query. At a decision point, running KDD enables a query to get up-to-date knowledge while incurring a cost of running KDD. On the other hand, not running KDD saves a cost of running KDD while making a query suffer from knowledge loss brought in by new data. Hence, the action space of the model is defined to be a binary set $A = \{0,1\}$, where 0 denotes not running KDD and 1 denotes running KDD.

The system state at decision point m is represented by a vector $s_m = (q_m^a, l_m)$, where q_m^a denotes the type of the query arrived at decision point m and l_m denotes the amount of knowledge loss at decision point m. The set of all feasible queries at decision point m is $\{q_i\}, i = 1,2,\cdots,n$. According to the independent Poisson arrival assumption for queries, the probability, $P\{q_m^a = q_i\}$, that type i query arrives at decision point m, is $\lambda_{q_i} \Big/ \sum_{j=1}^{n} \lambda_{q_j}$.

Let r_m be the latest decision point with action 1 (i.e., running KDD) before decision point m. l_m is determined by d_m, the number of units (e.g., blocks) of new data arrived between decision points r_m and m, and l, the expected amount of knowledge loss brought in by a unit of new data.

$$l_m = d_m l \tag{2.1}$$

The random nature of d_m makes the model intractable as it is impractical to solve the model for every possible value of d_m. To make the model tractable, we substitute d_m in (2.1) with its expected value $\lambda_d (m - r_m) \Big/ \sum_{j=1}^{n} \lambda_{q_j}$, and we have,

$$l_m = (m - r_m)\lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j} \tag{2.2}$$

According to (2.2), the set of all possible l_m at decision point m is L_m, where $L_m = \{\lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j}, 2\lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j}, \cdots, m\lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j}\}$. In L_m, $\lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j}$ indicates that the latest decision point with action 1 before decision point m is decision point $(m-1)$ and $m\lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j}$ indicates that KDD has never been run before decision point m. In summary, the state space S_m at decision point m can be defined as:

$$S_m = \{s_m \,|\, s_m = (q_m^a, l_m), \text{where } q_m^a \in \{q_i\} \text{ and } l_m \in L_m\} \tag{2.3}$$

Let $a_m \in A$ be the action chosen at decision point m. One step transition from state $s_m = (q_m^a, l_m)$ at decision point m to state $s_{m+1} = (q_{m+1}^a, l_{m+1})$ at decision point $(m+1)$ under action a_m consists of two stages. The first stage is the transition from l_m to l_{m+1}. Given l_m and a_m, there is only one possible amount of knowledge loss at next decision point, l_{a_m}, where,

$$l_{a_m} = \begin{cases} l_m + \lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j} & \text{if } a_m = 0 \\ \lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j} & \text{if } a_m = 1 \end{cases} \tag{2.4}$$

Thus, the transition probability $P^{a_m}_{l_m l_{m+1}}$ from l_m to l_{m+1} under action a_m is,

$$P^{a_m}_{l_m l_{m+1}} = \begin{cases} 1 & \text{if } l_m = l_{a_m} \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

The second stage is the transition from q^a_m to q^a_{m+1}, which results from the arrival of query q^a_{m+1}. Hence, the transition probability of the second stage is $P\{q^a_{m+1} = q_i\}$, where $i = 1,2,\cdots,n$. The second stage transition is independent of the first stage transition as query arrival is independent of new data arrival and the action chosen. As a result, one step transition probability $P^{a_m}_{s_m s_{m+1}}$ from state s_m to s_{m+1} under action a_m is determined as the product of $P^{a_m}_{l_m l_{m+1}}$ and $P\{q^a_{m+1} = q_i\}$,

$$P^{a_m}_{s_m s_{m+1}} = \begin{cases} P\{q^a_{m+1} = q_i\} & \text{if } l_m = l_{a_m} \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

where $P\{q^a_{m+1} = q_i\} = \lambda_{q_i} \Big/ \sum_{j=1}^{n} \lambda_{q_j}$.

The system cost $c(s_m, a_m)$ incurred at decision point m is,

$$c(s_m, a_m) = \begin{cases} c_m^l & \text{if } a_m = 0 \\ c_m^k & \text{if } a_m = 1 \end{cases} \tag{2.7}$$

where c_m^l denotes the cost of knowledge loss at decision point m and c_m^k denotes the cost of running KDD at decision point m. For different types of queries in $\{q_i\}, i = 1,2,\cdots,n$, let the cost of per unit knowledge loss be c_{q_i} respectively.

$$c_m^l = c_{q_i} l_m \qquad \text{if } q_m^a = q_i, i = 1,2,\cdots,n \tag{2.8}$$

Substituting l_m in (2.8) with (2.2), we have,

$$c_m^l = (m - r_m)\lambda_d l c_{q_i} \Big/ \sum_{j=1}^{n} \lambda_{q_j} \qquad \text{if } q_m^a = q_i, i = 1,2,\cdots,n \tag{2.9}$$

Although the computational cost of data mining is proportional to the size of the mined data, the cost of data pre-processing and pattern post-processing, which usually involves people, typically does not vary with data volume and the latter dominates in the cost of running KDD. Hence, we model the expected cost c_m^k of running KDD at decision point m as a constant C_k. Therefore, the system cost $c(s_m, a_m)$ incurred at decision point m becomes,

$$c(s_m, a_m) = \begin{cases} (m - r_m)\lambda_d lc_{q_i} \Big/ \sum_{j=1}^{n} \lambda_{q_j} & \text{if } a_m = 0 \text{ and } q_m^a = q_i, i = 1,2,\cdots,n \\ C_k & \text{if } a_m = 1 \end{cases} \quad (2.10)$$

The expected number of decision points over a relative long time horizon, M, is $T\sum_{j=1}^{n} \lambda_{q_j}$, where T is the duration of the time horizon. A refreshing policy π can be defined as $\pi = (\delta_1, \delta_2, \cdots, \delta_m, \cdots, \delta_M)$, where δ_m is the decision rule at decision point m such that $a_m = \delta_m(s_m)$. The expected total system cost over the time horizon under refreshing policy π, EC_π, can be expressed as,

$$EC_\pi = E\left\{\sum_{i=1}^{M} [\delta_i(s_i)c_i^k + (1 - \delta_i(s_i))c_i^l]\right\} \quad (2.11)$$

The optimal knowledge refreshing policy, π^*, minimizes expected total system cost over the time horizon such that $EC_{\pi^*} = \min_{\pi \in \Pi} EC_\pi$, where policy space Π is the set of all feasible knowledge refreshing policies.

2.2 Solution Methods

2.3.1 The Global Optimal Knowledge Refreshing Policy

An exhaustive enumeration of all knowledge refreshing policies in Π to search for the optimal one requires a time complexity $O(2^M)$, where M is the expected number of

decision points over the time horizon, as there are 2^M different knowledge refreshing policies in Π. The exponential nature of the solution discourages its application when M is large. A dynamic programming solution recursively evaluates the individual knowledge refreshing decisions over a time horizon in their reverse time order (Bellman 1957). Let $v_{s_m}(m)$ be the optimal expected total system cost from decision point m to the end of the time horizon with current state s_m, where $1 \le m \le M$. $v_{s_m}(m)$ can be calculated as,

$$v_{s_m}(m) = \min_{a_m \in A}\{c(s_m, a_m) + \sum_{s_{m+1} \in S_{m+1}} P^{a_m}_{s_m s_{m+1}} v_{s_{m+1}}(m+1)\} \tag{2.12}$$

Let $v(0)$ be the optimal expected total system cost from the start to the end of the time horizon.

$$v(0) = \sum_{s_1 \in S_1} P_{s_1} v_{s_1}(1) \tag{2.13}$$

In (2.13),

$$S_1 = \{s_1 \big| s_1 = (q_1^a, l_1), \text{where } q_1^a \in \{q_i\} \text{ and } l_1 = \lambda_d l \Big/ \sum_{j=1}^{n} \lambda_{q_j}\}$$

$$P_{s_1} = \lambda_{q_i} \Big/ \sum_{j=1}^{n} \lambda_{q_j} \qquad \text{if } s_1 = (q_i, l_1), \text{for } i = 1,2,\cdots,n$$

Applying the dynamic programming solution, $v(0)$ can be solved backwards from decision point M. At the same time, EC_{π^*} and π^* are obtained, where $EC_{\pi^*} = v(0)$. The decision rules in the optimal knowledge refreshing policy π^* are IF-THEN rules. These IF-THEN rules have a general format of "IF the system state at the current decision point is ..., THEN the optimal action is ...". Instead of searching the policy space, the dynamic programming solution sweeps through all possible system states at each decision point to obtain π^*. According to (2.3), the cardinality of S_m, the state space at decision point m, is nm. Hence, the time complexity of the dynamic programming solution is $O(n\sum_{j=1}^{M} j)$, which is $O(M^2)$ given that n is negligible compared to M.

2.3.2 The Fixed Optimal Knowledge Refreshing Policy

Although tractable, the dynamic programming solution is still computational expensive for large M. Moreover, the number of optimal decision rules generated by the dynamic programming solution has the order of $O(M^2)$ and searching for the optimal decision rule at a decision point makes the solution even more expensive. A more computationally efficient solution can be obtained by reducing police space from Π to Π_f. Π_f contains only policies refreshing knowledge every fixed number of queries/decision points, namely fixed knowledge refreshing policy space. Past research (Young 1974, Chandy et al. 1975) has proposed theorems for determining optimal fixed time intervals between checkpoints in Operation Systems and Database Management Systems. Motivated by

such research, we introduce Lemma 1 to determine the fixed optimal knowledge refreshing policy, π_f^*, such that $EC_{\pi_f^*} = \min_{\pi_f \in \Pi_f} EC_{\pi_f}$, where π_f is a fixed knowledge refreshing policy.

Lemma 1. The fixed optimal knowledge refreshing policy π_f^* is a policy that runs KDD every Q^* queries/decision points (i.e., runs KDD at decision points $Q^*, 2Q^*, \cdots$) where

$$Q^* = \sqrt{\frac{2C_k \sum_{j=1}^{n} \lambda_{q_j}}{I\bar{c}_q \lambda_d}} \tag{2.14}$$

and $\bar{c}_q = \dfrac{\sum_{j=1}^{n} \lambda_{q_j} c_{q_j}}{\sum_{j=1}^{n} \lambda_{q_j}}$.

Proof. Following a knowledge refreshing policy in Π_f, KDD is run every Q decision points. The expected cost of running KDD over the time horizon, C_{KDD}, is

$$C_{KDD} = C_k \frac{M}{Q} \tag{2.15}$$

The time horizon can be divided into $\dfrac{M}{Q}$ periods with each period including Q queries. Using (2.8), the expected cost of knowledge loss at each period, $C_{KL/P}$, is

$$C_{KL/P} = \frac{\lambda_d}{\sum_{j=1}^{n} \lambda_{q_j}} l \sum_{k=1}^{Q-1} k \frac{\sum_{j=1}^{n} \lambda_{q_j} c_{q_j}}{\sum_{j=1}^{n} \lambda_{q_j}}$$

Let $\bar{c}_q = \frac{\sum_{j=1}^{n} \lambda_{q_j} c_{q_j}}{\sum_{j=1}^{n} \lambda_{q_j}}$, where $\bar{c}_q$ is the expected cost of per unit knowledge loss. We have,

$$C_{KL/P} = \frac{\lambda_d}{\sum_{j=1}^{n} \lambda_{q_j}} l \frac{Q(Q-1)}{2} \bar{c}_q$$

The expected cost of knowledge loss over the time horizon, C_{KL}, is

$$C_{KL} = C_{KL/P} \frac{M}{Q} = \frac{\lambda_d}{\sum_{j=1}^{n} \lambda_{q_j}} l \frac{M(Q-1)}{2} \bar{c}_q \tag{2.16}$$

Adding (2.15) and (2.16), the expected system cost over the time horizon, *EC*, is

$$EC = C_k \frac{M}{Q} + \frac{\lambda_d}{\sum_{j=1}^{n} \lambda_{q_j}} l \frac{M(Q-1)}{2} \bar{c}_q \tag{2.17}$$

Since $\frac{\partial^2 (EC)}{\partial Q^2} > 0$, solving $\frac{\partial (EC)}{\partial Q} = 0$ gives (2.14) that minimizes expected system cost for all fixed knowledge refreshing policies.

Lemma 1 indicates that the fixed optimal knowledge refreshing policy π_f^* is determined by the following factors: C_k, the cost of running KDD, λ_{q_i}, $i = 1,2,\cdots n$, the arrival rates of different types of queries to the knowledge base, λ_d, the arrival rate of new data and $\overline{lc}_q$, the expected cost of knowledge loss brought in by a unit of new data. The time complexity of inducing π_f^* is $O(1)$, while the time complexity of determining π^* is $O(M^2)$. According to Lemma 1, all the decisions in π_f^* can be pre-determined before the start of the time horizon using (2.14), while the decisions in π^* depend on current system states. Hence, the implementation of π_f^* is more computational efficient than the implementation of π^*. However, as shown in Lemma 2, the expected system cost under π_f^* is greater than or equal to the expected system cost under π^*.

Lemma 2. $$EC_{\pi_f^*} \geq EC_{\pi^*} \tag{2.18}$$

Proof. Since $\Pi_f \subset \prod$, we have,

$$EC_{\pi_f} \geq EC_{\pi^*} \qquad \forall \pi_f \in \Pi_f$$

We also have, $\pi_f^* \in \Pi_f$. Hence, (2.18) follows.

Due to the computational and implementation efficiency of the fixed optimal knowledge refreshing policy π_f^*, it is highly critical in knowledge refreshing to determine: (1) the

conditions under which $EC_{\pi_f^*}$ is equivalent to EC_{π^*}, and (2) the magnitude of system cost saving generated by implementing π^*, compared with implementing π_f^*. The first question will be addressed by Lemma 3 and the second question is numerically explored in Section 2.3.

Lemma 3. $EC_{\pi_f^*} = EC_{\pi^*}$, if one of the following two conditions is satisfied:

(1) $$\frac{C_k \sum_{j=1}^{n} \lambda_{q_j}}{l \min(c_q) \lambda_d} < 1$$

(2) $$\frac{C_k \sum_{j=1}^{n} \lambda_{q_j}}{l \max(c_q) \lambda_d} > \frac{M(M+1)}{2}$$

In (1) and (2), $\min(c_q)$ and $\max(c_q)$ are the minimum and the maximum cost among c_{q_i}, for $i = 1,2,\cdots n$.

Proof. Under condition (1), we get,

$$C_k < \frac{\lambda_d}{\sum_{j=1}^{n} \lambda_{q_j}} l \min(c_q) \tag{2.19}$$

According to (2.2), the minimum amount of knowledge loss at decision point m, for $1 \le m \le M$, is resulted if $r_m = m-1$, which indicates that the latest decision point with action 1 before decision point m is decision point $(m-1)$. Substituting r_m in (2.2) with $m-1$, we get $\min(l_m)$, the minimum amount of knowledge loss at decision point m, for $1 \le m \le M$,

$$\min(l_m) = \frac{\lambda_d}{\sum_{j=1}^{n} \lambda_{q_j}} l \tag{2.20}$$

Multiplying (2.20) with $\min(c_q)$, we get $\min(c_m^l)$, the minimum cost of knowledge loss at decision point m, for $1 \le m \le M$,

$$\min(c_m^l) = \frac{\lambda_d}{\sum_{j=1}^{n} \lambda_{q_j}} l \min(c_q) \tag{2.21}$$

According to (2.19) and (2.21), we have $C_k < \min(c_m^l)$, for $1 \le m \le M$, which means that, at any decision point over the time horizon, the cost of running KDD is less than the minimum cost of knowledge loss at the decision point. Hence, under condition (1), the global optimal knowledge refreshing policy in π^* requires KDD to be run whenever there is a query to the knowledge base.

Under condition (1), we also have,

$$\frac{C_k \sum_{j=1}^{n} \lambda_{q_j}}{l\bar{c}_q \lambda_d} < 1 \tag{2.22}$$

because $\bar{c}_q \geq \min(c_q)$. Applying (2.12) to (2.14), we get $Q^* < \sqrt{2}$. Hence, under condition (1), the fixed optimal knowledge refreshing policy π_f^* also requires KDD to be run whenever there is a query to the knowledge base. Therefore, $EC_{\pi_f^*} = EC_{\pi^*}$, if condition (1) is satisfied.

Under condition (2), we get,

$$C_k > \frac{\lambda_d}{\sum_{j=1}^{n} \lambda_{q_j}} l \max(c_q) \frac{M(M+1)}{2} \tag{2.23}$$

The right side of (2.23) represents the maximum cost of knowledge loss over the time horizon if no KDD has been run over the time horizon. Hence, under condition (2), the global optimal knowledge refreshing policy π^* requires no KDD to be run over the time horizon.

Under condition (2), we also have,

$$\frac{C_k \sum_{j=1}^{n} \lambda_{q_j}}{l\bar{c}_q \lambda_d} > \frac{M(M+1)}{2} \tag{2.24}$$

because $\bar{c}_q \leq \max(c_q)$. Applying (2.24) to (2.14), we get $Q^* > M$. Hence, under condition (2), the fixed optimal knowledge refreshing policy π_f^* also requires no KDD to be run over the time horizon. Therefore, $EC_{\pi_f^*} = EC_{\pi^*}$, if condition (2) is satisfied.

Lemma 3 gives two conditions under which $EC_{\pi_f^*}$ is equivalent to EC_{π^*}. For the situations between the two extreme conditions given in Lemma 3, the magnitude of cost saving between EC_{π^*} and $EC_{\pi_f^*}$ decreases when the situation is closer to one of the extreme conditions, but increases when the situation moves farther from the extreme conditions. We will explore the magnitude of cost saving between EC_{π^*} and $EC_{\pi_f^*}$ numerically in Section 2.3.3.

2.3 Numerical Analysis

The objective of numerical analysis is two-fold. First, we will evaluate the robustness of the model with respect to its assumptions. Second, we will examine how the system parameters impact the magnitude of cost saving between EC_{π^*} and $EC_{\pi_f^*}$.

2.3.1 Simulation Environment

In the simulation environment, there are three distinct types of queries: q_1, q_2 and q_3. q_1, q_2 and q_3 are simulated as Poisson processes with arrival rate 0.6, 0.06 and 0.015 respectively. The cost of per unit knowledge loss for q_1, q_2 and q_3 are 0.1, 1.6 and 2.0 respectively. The arrival of new data is also simulated as a Poisson process with intensity 10. The amount of knowledge loss brought in by a unit of new data is simulated as a uniform distributed random variable with mean 0.1 unit. And the cost of running KDD is a constant with value 40. Values of major system parameters are summarized in Table 2.2.

Values of major system parameters are set to reflect real world situations. We use one month web log collected from an non-profit organization to estimate $\lambda_{q_1}, \lambda_{q_2}, \lambda_{q_3}$ and λ_d. In the web log, there are on average 12833 blocks of data arrived per hour (λ_d) and 875 visits arrived per hour. By assuming the ratio between $\lambda_{q_1}, \lambda_{q_2}$ and λ_{q_3} as 40:4:1, we have the values of $\lambda_{q_1}, \lambda_{q_2}$ and λ_{q_3} as 777 (i.e., $875 \times 40/45$), 77 and 19 respectively. Scaling

down the values of $\lambda_{q_1}, \lambda_{q_2}, \lambda_{q_3}$ and λ_d by multiplying $\frac{1}{1283}$, we have the values of $\lambda_{q_1}, \lambda_{q_2}, \lambda_{q_3}$ and λ_d as listed in Table 2.2. In real world, query type with higher arrive rate usually incurs lower cost of knowledge loss. For example, queries to support strategic decision making usually have a lower arrival rate than queries to support operational decision making; while the former incurs higher cost of knowledge loss than the latter. Thinking of a knowledge based Business-to-Consumer web portal, window shoppers usually have a higher arrival rate than buyers; while the former generates lower revenue than the latter. Hence, as shown in Table 2.2, we set $\lambda_{q_1} > \lambda_{q_2} > \lambda_{q_3}$ and $c_{q_1} < c_{q_2} < c_{q_3}$. The values of $c_{q_1}, c_{q_2}, c_{q_3}$ and C_k are set to be 0.1, 16., 2.0 and 40 respectively.

Table 2.2: Values of the Simulation Parameters

Parameter	Value
The arrival rate of q_1, λ_{q_1}	0.6
The arrival rate of q_2 λ_{q_2}	0.06
The arrival rate of q_3, λ_{q_3}	0.015
The arrival rate of new data, λ_d	10
Mean of the knowledge loss brought in by a unit of new data, l	0.1
The cost of per unit knowledge loss for q_1, c_{q_1}	0.1
The cost of per unit knowledge loss for q_2 c_{q_2}	1.6
The cost of per unit knowledge loss for q_3, c_{q_3}	2.0
The cost of running KDD, C_k	40

All of the simulation results are collected from experiments with 500 simulation runs, yielding small confidence intervals for the simulation results with confidence level 90%

(Molloy 1989). Before running simulation experiments, the Markov decision process model proposed in Section 2.1 is solved using the dynamic programming method, resulting in the global optimal knowledge refreshing policy π^*, and the analytical result of EC_{π^*}. For the actual implementation of π^* in the simulation environment, the optimal decision rule for any query generated between the time interval $(\frac{mT}{M}, \frac{(m+1)T}{M}]$ is set to be δ_m^*, where δ_m^* is the optimal decision rule for the m-th query under π^*. To check the soundness of the π^* implementation, we compare analytical results of EC_{π^*} with simulation results of EC_{π^*}. Table 2.3 summarizes the results of the comparison, in which the length of the time horizon T is increased from 400 to 600 with incremental step size 50. The simulation error in Table 2.3 is the absolute difference between the analytical result of EC_{π^*} and the simulation result of EC_{π^*} divided by the analytical result of EC_{π^*}, expressed in percentage. As shown in Table 2.3, the small simulation error not only approves the proposed scheme of implementing π^* in the simulation environment but also shows the robustness of using the expected values of d_m in the analytical model.

Table 2.3: Comparison Between Simulation and Analytical Results of EC_{π^*}

T	Analytical result of EC_{π^*}	Simulation result of EC_{π^*}	Simulation error (%)
400	1153.49	1108.557656	3.89
450	1296.93	1265.449724	2.42
500	1444.717	1398.226498	3.21
550	1592.503	1530.530327	3.89
600	1735.943	1668.185278	3.90

2.3.2 Robustness Analysis

2.3.2.1 Robustness Analysis With Respect to the Poisson Arrival Assumptions

Both new data interarrival times and query interarrival times in the model are assumed to be exponentially distributed to obtain analytical tractability of the problem. To determine the impact of the Poisson arrival assumption on the precision of the model and its associated optimal solutions, simulation experiments have been performed with non-exponential distributions for new data and query interarrival times. Uniform and Erlang distributions are used in the experiments as they have different shapes and variances from the exponential distribution. In the experiments, the global optimal knowledge refreshing policy π^*, is solved using the dynamic programming method under the Poisson arrival assumptions for both new data and queries. Using the scheme described in Section 2.3.1, π^* is then implemented in the three simulation experiments with exponential, uniform and Erlang interarrival times for both new data and queries respectively, while keeping the values of other system parameters same as those given in Table 2.2. Table 2.4 summarizes the results of the experiments, in which the length of the time horizon T is increased from 400 to 600 with incremental step size 50. In Table 2.4, the simulation difference is the absolute difference between the simulation result of EC_{π^*} under the non-Poisson arrival assumption and the simulation result of EC_{π^*} under the Poisson arrival assumption divided by the latter, expressed in percentage. The small simulation

difference indicates that the global optimal knowledge refreshing policy π^* is not sensitive to the interarrival times distributions of new data and queries.

Table 2.4: Simulation Results For The Robustness Analysis With Respect To The Impact of Interarrival Times Assumptions On π^*

	Exponential arrival	Uniform arrival		Erlang arrival	
T	EC_{π^*}	EC_{π^*}	Simulation difference(%)	EC_{π^*}	Simulation difference(%)
400	1108.557656	1107.531775	0.09	1119.177993	0.96
450	1265.449724	1251.570541	1.09	1275.402318	0.79
500	1398.226498	1393.219351	0.36	1404.347574	0.44
550	1530.530327	1537.677213	0.47	1560.881348	1.98
600	1668.185278	1671.274484	0.19	1706.887144	2.32

Simulation experiments have also been performed to check the impact of the interarrival times distributions on the fixed optimal knowledge refreshing policy π_f^*. In the experiments, T is set to be 600. According to (2.12), π_f^* under the Poisson arrival assumptions is to run KDD every 14 queries and the analytical result of $EC_{\pi_f^*}$ is 2231.81, using the values of the system parameters given in Table 2.2. Simulation results of the three simulation experiments with exponential, uniform and Erlang interarrival times are summarized in Table 2.5. In Table 2.5, we find that: (1) EC_{π_f} under different interarrival times assumptions become optimal at the same value of Q, which is also same as the analytical optimal value of Q; (2) simulation results of $EC_{\pi_f^*}$ under different interarrival times assumptions are close to the analytical result of $EC_{\pi_f^*}$; (3) the simulation

difference between EC_{π_f} under different interarrival times assumptions is small. The simulation results reveal that the interarrival times distribution has little impact on π_f^*.

Table 2.5: Simulation Results For The Robustness Analysis With Respect To The Impact of Interarrival Times Assumptions On π_f^* (T=600)

	Exponential arrival	Uniform arrival		Erlang arrival	
Q	EC_{π_f}	EC_{π_f}	Simulation difference(%)	EC_{π_f}	Simulation difference(%)
4	4214.037	4229.593	0.36	4249.254	0.84
8	2547.33	2547.255	0.002	2552.437	0.20
12	2215.447	2202.738	0.57	2214.582	0.04
14	2190.836	2178.303	0.57	2177.38	0.61
16	2198.048	2192.112	0.27	2201.592	0.16
20	2306.59	2323.313	0.72	2309.693	0.13
24	2517.35	2510.36	0.27	2497.985	0.77

2.3.2.2 Robustness Analysis With Respect to Using the Expected Knowledge Loss

Expected amount of knowledge loss is used in the proposed Markov decision process model. In real world, the amount of knowledge loss brought in by a unit of new data could follow a variety of distributions. To verify the robustness of the model with respect to using expected amount of knowledge loss, we need to check how the distribution of knowledge loss impact the optimal knowledge refreshing policies π^* and π_f^*.

With the length of the time horizon T increased from 400 to 600 with incremental step size 50, its corresponding π^* and EC_{π^*} are calculated by applying the dynamic

programming method to the model described in Section 2.1. π^* are then implemented in the simulation environment described in Section 2.3.1 with uniformly and exponentially distributed knowledge loss respectively. Analytical and simulation results of EC_{π^*} are summarized in Table 2.6. Both the simulation error between analytical and simulation results of EC_{π^*} and the simulation difference between two simulation results under different knowledge loss distributions are small. The simulation results indicate that (1) π^* is not sensitive to the distribution of knowledge loss; (2) using the expected value of knowledge loss has little impact on π^*.

Table 2.6: Simulation Results For The Robustness Analysis With Respect To The Impact of Knowledge Loss Distribution On π^*

T	Analytical result of EC_{π^*}	Simulation result of EC_{π^*}				
		Uniform	Simulation error (%)	Exponential	Simulation error (%)	Simulation Difference (%)
400	1153.49	1108.55	3.89	1112.41	3.56	0.35
450	1296.93	1265.44	2.42	1245.82	3.94	1.55
500	1444.717	1398.22	3.21	1398.80	3.17	0.04
550	1592.503	1530.53	3.89	1513.13	4.98	1.37
600	1735.943	1668.18	3.90	1675.74	3.46	0.45

Setting T to be 600 and using the values of the system parameters given in Table 2.2, the analytical result of Q^* is 14 and the analytical result of $EC_{\pi_f^*}$ is 2231.81, according to Lemma 1. Simulation results of the experiments with uniformly and exponentially distributed knowledge loss are summarized in Table 2.7. In Table 2.7, we find that: (1)

EC_{π_f} under different knowledge loss distributions become optimal at the same value of Q, which is also same as the analytical optimal value of Q; (2) simulation results of EC_{π_f} under different knowledge loss distributions are close to the analytical result of EC_{π_f}; (3) the simulation difference between EC_{π_f} under different knowledge loss distributions is small. The simulation results reveal that using the expected value of knowledge loss has little impact on π_f^*.

Table 2.7: Simulation Results For The Robustness Analysis With Respect To The Impact of Knowledge Loss Distribution On π_f^* (T=600)

	Uniform	Exponential	
Q	EC_{π_f}	EC_{π_f}	Simulation difference(%)
4	4214.037	4239.24	0.60
8	2547.33	2552.52	0.20
12	2215.447	2211.08	0.19
14	2190.836	2184.59	0.29
16	2198.048	2205.81	0.35
20	2306.59	2324.79	0.79
24	2517.35	2529.29	0.47

2.3.3 Sensitivity Analysis

In this section, we analyze numerically the impact of system parameters on the cost saving generated by implementing π^*, instead of π_f^*. The following system parameters are evaluated in this section:

(1) the arrival rate of new data, λ_d;

(2) the cost of running KDD, C_k;

(3) the expected knowledge loss brought in by a unit of new data, l;

(4) the arrival rates of queries.

2.3.3.1 The Impact of the Arrival Rate of New Data λ_d

According to Lemma 3, if the arrival rate of new data λ_d is large enough that condition (1) is satisfied, then both π^* and π_f^* require to run KDD for every query to the knowledge base. As a result, EC_{π^*} equals to $EC_{\pi_f^*}$. On the other hand, if the arrival rate of new data λ_d is small enough that condition (2) is satisfied, EC_{π^*} is also equivalent to $EC_{\pi_f^*}$. For the values of λ_d between the two boundary values, performance improvement between EC_{π^*} and $EC_{\pi_f^*}$ increases as the values are farther from the two boundary values while decreases as the values move closer to one of the boundary values. To verify the performance impact of λ_d, we increase the value of λ_d from 0.01 to 2700, while keep the values of the rest of the system parameters fixed. In the experiment, the duration of the time horizon is set to be 600 and the two boundary values of λ_d are 2700 and 0.01. Simulation results summarized in Table 2.8 and Figure 2.2 are consistent with Lemma 3. In the table, performance improvement is the absolute difference between the simulation

result of EC_{π^*} and the simulation result of $EC_{\pi_f^*}$ divided by the latter, expressed in percentage. However, Lemma 3 cannot help determine:

(1) the shape of the curve in Figure 2.2;

(2) and the condition at which the difference between EC_{π^*} and $EC_{\pi_f^*}$ is maximized.

We leave these two research questions as a future research direction of the dissertation.

Table 2.8: Performance Improvement Between EC_{π^*} And $EC_{\pi_f^*}$ With Respect To The Impact of The Arrival Rate of New Data λ_d (T=600)

λ_d	**Simulation result of** $EC_{\pi_f^*}$	**Simulation result of** EC_{π^*}	**Performance Improvement (%)**
2700	16209	16160	0.30
1280	15603.98	12154.96	28.37
320	10447.32	7354.48	42.05
80	5884.99	4174.87	40.96
20	3045.47	2259.23	34.80
0.01	31.43	31.04	1.24

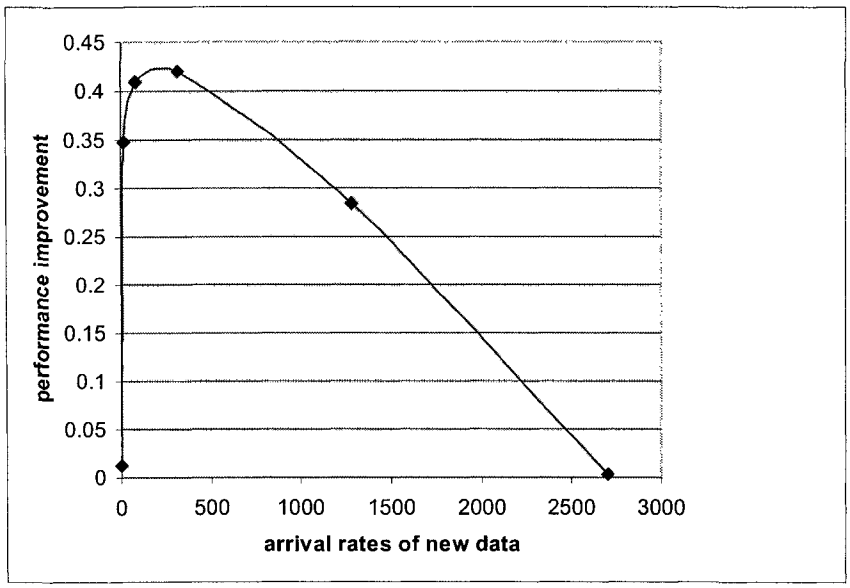


Figure 2.2: Performance Improvement Between EC_{π^*} And EC_{π_f} With Respect To The Impact of The Arrival Rate of New Data λ_d (T=600)

2.3.3.2 The Impact of the Cost of Running KDD C_k

In this experiment, we increase the cost of running KDD C_k from 0.14 to 34000, keeping the values of other system parameters fixed: the length of the time horizon being set to be 600 and the values of the rest system parameters being same as what described in Section 2.3.1. The simulation results are summarized in Table 2.9.

Table 2.9: Performance Improvement Between EC_{π^*} And $EC_{\pi_f^*}$ With Respect To The Impact Of The Cost of Running KDD C_k (T=600)

C_k	Simulation result of $EC_{\pi_f^*}$	Simulation result of EC_{π^*}	Performance Improvement (%)
0.14	56.285	56.172	0.202
0.3	120.492	85.751	40.513
2.5	483.948	334.924	44.495
20	1520.859	1126.595	34.996
160	4389.055	3804.131	15.376
34000	33093.680	33085.580	0.245

From Table 2.9, we can find that (1) $EC_{\pi_f^*}$ is close to EC_{π^*} when C_k is small (i.e., 0.14) or large (e.g., 34000) enough that one of the conditions in Lemma 3 is satisfied. The small performance improvements are due to simulation errors. (2) Performance improvement between EC_{π^*} and $EC_{\pi_f^*}$ becomes more apparent as C_k move farther from either 0.14 or 34000. And the performance improvement reaches its highest value when C_k is 2.5. Simulation results generated in this experiment are consistent with Lemma 3.

2.3.3.3 The Impact of the Expected Knowledge Loss *l*

To evaluate the impact of the expected knowledge loss on the performance improvement between EC_{π^*} and $EC_{\pi_f^*}$, we increase the expected knowledge loss brought in by a unit of new data from 0.0001 to 28, keeping the values of the rest system parameters fixed. Similarly, the duration of the time horizon is set to be 600 in this experiment. Simulation results of the experiment are summarized in Table 2.10.

Table 2.10: Performance Improvement Between EC_{π^*} And $EC_{\pi_f^*}$ With Respect To The Impact of The Expected Knowledge Loss l (T=600)

L	Simulation result of $EC_{\pi_f^*}$	Simulation result of EC_{π^*}	Performance Improvement (%)
28	16072.40	16033.33	0.24
16	16080.16	11751.49	36.84
4	10909.62	7491.58	45.62
1	6319.72	4344.34	45.47
0.1	2190.83	1668.18	31.33
0.01	695.46	623.40	11.55
0.0001	33.09	33.06	0.07

When the expected knowledge loss brought in by a unit of new data is large enough that condition (1) in Lemma 3 is satisfied, both π^* and π_f^* require to run KDD for every query to the knowledge base. Hence, we can see from the simulation results that EC_{π^*} equals to $EC_{\pi_f^*}$, except for the small simulation error. In contrast, when the expected knowledge loss brought in by a unit of new data is small enough that condition (2) in Lemma 3 is satisfied, both π^* and π_f^* forbid running KDD for every query to the knowledge base. As a result, EC_{π^*} also equals to $EC_{\pi_f^*}$ except for the small simulation error. For the values of the expected knowledge loss between 0.0001 and 28, performance improvement between EC_{π^*} and $EC_{\pi_f^*}$ increases as the values are farther from the two boundary values while decreases as the values move closer to one of the boundary values.

2.3.3.4 The Impact of the Arrival Rate of Queries

In this experiment, we increase the arrival rate of query q_1, λ_{q_1}, from 0.004 to 4, while keep the values of other system parameters fixed. The length of the time horizon is set to be 600 in this experiment. The simulation results are summarized in Table 2.11.

Table 2.11: Performance Improvement Between EC_{π^*} And $EC_{\pi_f^*}$ With Respect To The Impact of The Arrival Rate of Query q_1, λ_{q_1} (T=600)

λ_{q_1}	Simulation result of $EC_{\pi_f^*}$	Simulation result of EC_{π^*}	Performance Improvement (%)
4	3809.436	3572.905	6.62
1	2431.944	1996.15	21.83
0.25	1895.638	1422.423	33.26
0.0625	1632.786	1274.094	28.15
0.016	1493.106	1263.723	18.15
0.004	1405.339	1248.067	12.60

From Table 2.11, we can find that the performance improvement between EC_{π^*} and $EC_{\pi_f^*}$ reaches its maximum value when λ_{q_1} is 0.25. And the performance improvement decreases as the value of λ_{q_1} moves towards one of the boundary values of λ_{q_1} that make condition (1) or (2) in Lemma 3 satisfied. Similar trends can be found when varying λ_{q_2} and λ_{q_3}, as shown in Table 2.12 and Table 2.13.

Table 2.12: Performance Improvement Between EC_{π^*} And $EC_{\pi_f^*}$ With Respect To The Impact of The Arrival Rate of Query q_2, λ_{q_2} (T=600)

λ_{q_2}	Simulation result of $EC_{\pi_f^*}$	Simulation result of EC_{π^*}	Performance Improvement (%)
0.24	3479.245	2981.017	16.71
0.12	2687.065	2174.323	23.58
0.06	2182.968	1668.185	30.85
0.03	1879.604	1534.249	22.51
0.015	1703.096	1436.898	18.52

Table 2.13: Performance Improvement Between EC_{π^*} And $EC_{\pi_f^*}$ With Respect To The Impact of The Arrival Rate of Query q_3, λ_{q_3} (T=600)

λ_{q_3}	Simulation result of $EC_{\pi_f^*}$	Simulation result of EC_{π^*}	Performance Improvement (%)
0.025	2297.085	1844.109	24.56
0.02	2233.939	1774.85	25.86
0.015	2182.968	1668.185	30.85
0.01	2127.339	1697.507	25.32
0.005	2063.931	1659.575	24.36

2.4 Conclusion

In this chapter, we develop a Markov decision process model of knowledge refreshing. Two optimal knowledge refreshing policies – π^*, the global optimal knowledge refreshing policy, and π_f^*, the fixed optimal knowledge refreshing policy, are derived from the model. Relationships between π^* and π_f^* are studied both analytically and

numerically. Research results from this chapter provide the following insights for knowledge refreshing heuristics development:

(1) The knowledge refreshing model developed in Section 2.1 provides structural guidelines to analyze real world knowledge refreshing problems;

(2) The optimal knowledge refreshing policies derived in Section 2.2 provides bases for developing knowledge refreshing heuristics;

Before discussing implementation procedure for optimal knowledge refreshing policies and proposing heuristics for knowledge refreshing, we introduce real world knowledge refreshing requirements using two real world KDD applications: web mining based web portal design described in Chapter 3 and data mining based caching approach illustrated in Chapter4. We then summarize the knowledge refreshing requirements in the two applications, propose implementation procedure for optimal knowledge refreshing policies and develop heuristics based on insights generated in this chapter.

CHAPTER 3: LINKSELECTOR: A WEB MINING APPROACH TO HYPERLINK SELECTION FOR WEB PORTALS

In this chapter, we introduce a real world KDD application – a web mining based hyperlink selection approach for web portals. Besides contributions to the area of intelligent web portal design, the application described in this chapter also provides a real world environment to study the knowledge refreshing problem. The majority of the chapter describes LinkSelector – a web mining based hyperlink selection approach for web portals. We briefly summarize knowledge refreshing requirements for LinkSelector at the end of the chapter. Detailed knowledge refreshing requirements for LinkSelector are illustrated and addressed in Chapter 5, based on the analytical model and its optimal solutions developed in Chapter 2.

3.1 Introduction

As the size and complexity of websites expands dramatically, it has become increasingly challenging to design websites on which web surfers can easily find the information they seek. To address this challenge, we formally define a research problem in this area, hyperlink selection, and present a web mining based approach, LinkSelector, as a solution.

There are two dominant ways through which web surfers find the information they seek (Chakrabarti 2000): using search engines and clicking on hyperlinks. Research on the

former is concerned with improving recall and precision (Lawrence and Giles 1998;1999; Chakrabarti 2000) of search engines. Our research, however, concentrates on improving the efficiency of the second way of web information searching. As web surfers click on a group of hyperlinks to find the information they seek, placing appropriate hyperlinks in web pages is critical to improving their web information searching efficiency. In particular, this chapter focuses on placing appropriate hyperlinks in the portal page of a website, which is the entrance to a website.

The homepage of a website is one type of portal page. Homepages which guide users to locate the information they seek easily create a good first impression and attract more users, while homepages which make information searching difficult result in a bad first impression and corresponding user loss (Nielson and Wagner 1996). A default web portal is another type of portal page. Recently, web portals that serve as a personal entrances to websites have attracted more and more attention. Universities such as UCLA have built educational web portals (e.g., My UCLA, http://my.ucla.edu); corporations such as Yahoo! have developed commercial web portals (e.g., My Yahoo!, http://my.yahoo.com). For practical purposes, portal service providers (e.g., Yahoo!) provide portal users with a standard default web portal, which the users can personalize (e.g., add or remove hyperlinks from the default web portal). As the first version of a web portal encountered by portal users, the default web portal plays an important role in the success of a web portal. According to My Yahoo!, most users never customize their default web portals and a great deal of effort should go into the design of the default web portal (Manber et al. 2000).

A portal page consists of hyperlinks selected from a hyperlink pool, which is a set of hyperlinks pointing to top-level web pages[1]. Usually, the hyperlink pool of a website consists of hyperlinks listed in the site-index page or the site-directory page. As shown in Figure 3.1, hyperlinks in the portal page of the University of Arizona website (http://www.arizona.edu) are selected from its hyperlink pool. The hyperlink pool consists of hyperlinks in its site-index page (http://www.arizona.edu/index/webindex.shtml). Hyperlinks in the portal page of My Yahoo! (http://my.yahoo.com) are also selected from its hyperlink pool. The pool, in this case, consists of hyperlinks in its site-directory page.

[1] Web pages in a website are organized in a hierarchy in which a high level web page is an aggregation of its low level web pages (Nielson 1999). For example, the web page of faculty list is one level higher than its corresponding faculty homepages and it is an aggregation of its corresponding faculty homepages. For a university website, top level web pages include the web page of department list and the web page of computing resources etc..

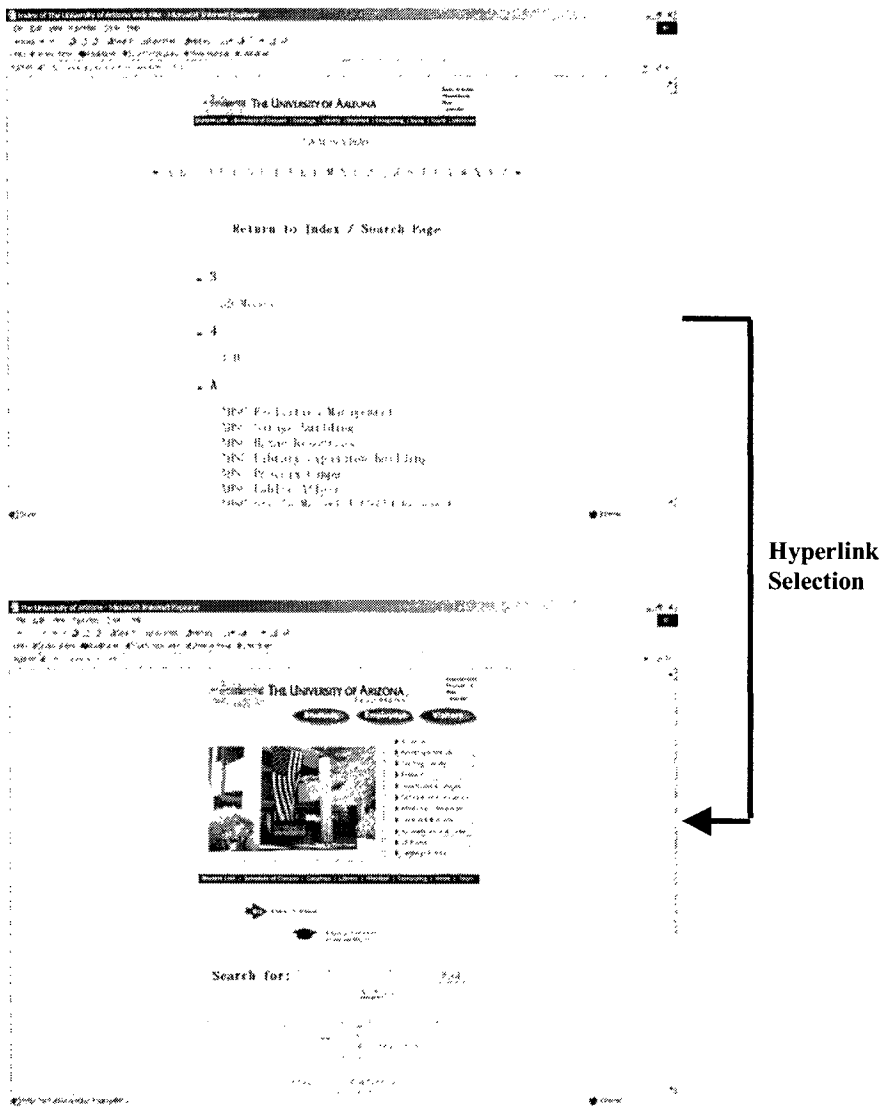


Figure 3.1: (a) The Hyperlink Pool (Top) And The Portal Page (Bottom) of The Website of The University of Arizona;

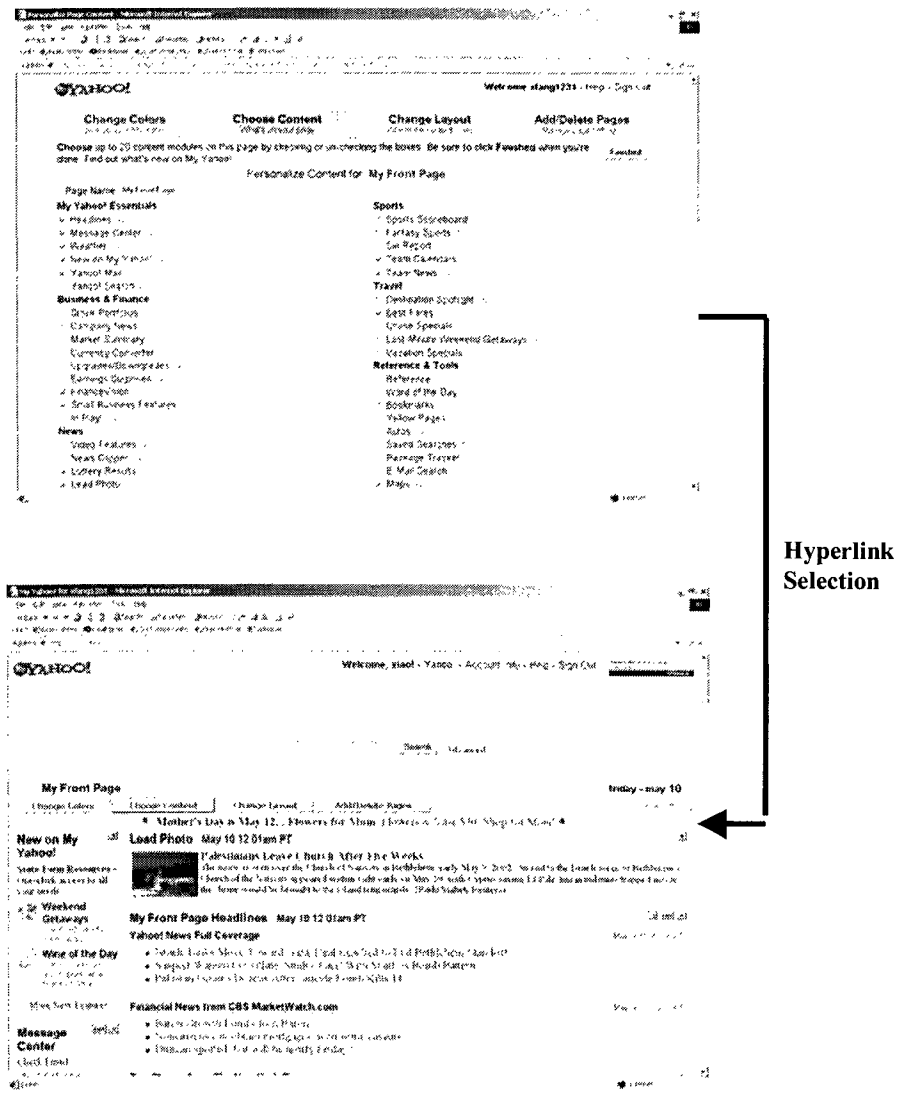


Figure 3.1 (b) The Hyperlink Pool (Top) and The Portal Page (Bottom) of My Yahoo!.

Given the web design principle that scrolling must be avoided in portal pages (Nielson 1999), a well-designed portal page normally contains several dozen (i.e., usually less than

4 dozen) hyperlinks[2]. However, the hyperlink pool of a typical website has at least several hundred hyperlinks. For example, the portal page of the University of Arizona website consists of 32 hyperlinks while the hyperlink pool has 743 hyperlinks. It is computationally too expensive to exhaust all combinations of several dozen hyperlinks from a hyperlink pool with several hundred hyperlinks and find the one that is the most efficient in guiding web surfers to find the information they seek. In this particular, for example, the number of combinations of selecting 32 hyperlinks from 743 hyperlinks is 1.44E+56 (i.e., C_{743}^{32}). Current practice of hyperlink selection relies on domain experts' (e.g., website designers) experiences. Obviously, such selection is subjective. In addition, it reflects only website designers' perspectives on what hyperlinks should be selected, not web surfers' perspectives. The second perspective should be emphasized as the purpose of hyperlink selection is to reduce web surfers' information searching efforts, not web designers'.

In comparison, our hyperlink selection method, LinkSelector, incorporates both patterns extracted from the structure of a website and those discovered from a web log, which records web surfers' behaviors of information searching. LinkSelector first employs web mining techniques (Kosala and Blockeel 2000) to extract the above-mentioned patterns and calculates preferences of hyperlinks and hyperlink sets defined in this chapter from these patterns. LinkSelector then selects hyperlinks from a given hyperlink pool by running the calculated preferences through a greedy algorithm developed in this chapter.

---

[2] Placing too many hyperlinks in a portal page will cause some hyperlinks to be visible only when scrolling down the window of the page. Unfortunately, according to Neilson's research (Nielson 1999), web surfers rarely scroll down the window of a portal page.

The rest of the chapter is organized as follows. We review related work in Section 3.2. In Section 3.3, we propose metrics to measure the quality of a portal page and formally define the hyperlink selection problem. A web mining based approach for hyperlink selection, LinkSelector, is presented in Section 3.4. In Section 3.5, we evaluate the performance of LinkSelector using data obtained from the University of Arizona website and the metrics proposed in Section 3.3. We conclude the chapter in Section 3.6.

3.2 Related Work

In this section, we first review works on web mining on which LinkSelector is based. In (Cooley et al. 1997), web mining is defined as the process of discovering and analyzing useful information from the Web. A good survey on web mining research can be found in (Kosala and Blockeel 2000). Srivastava et al. classified web data into content, structure and usage (Srivastava et al 2000):

- Content is the data in web pages. It usually consists of texts and graphics.

- Structure is the data describing the organization of the Web, such as hyperlinks.

- Usage is the data that describe web surfers' information searching behaviors. Web usage data can be found in web logs.

For different types of web data, corresponding web mining methods are developed. Web content mining is the process of automatically retrieving, filtering and categorizing web

documents, a good survey on which can be found in (Chakrabarti 2000). As it typically makes use of only texts on web pages, valuable information implicitly contained in hyperlinks is overlooked. Web structure mining (Chakrabarti et al. 1999) infers useful patterns from the Web's link topology to help retrieve high quality web pages. HITS and PageRank are two widely used web structure mining algorithms. In the HITS algorithm (Kleinberg 1998), authority pages were defined as high-quality pages related to a particular topic. Hub pages were those that were not necessarily authorities themselves but provided pointers to other authority pages. HITS calculated hub weights and authority weights of web pages using an iterative procedure and returned web pages with the highest hub weights or authority weights. The PageRank algorithm (Brin and Page 1998) weighted each in-link to a page proportionally to the quality of the page containing the in-link. A web page had a high PageRank score if it was linked from many other pages, and the scores were even higher if these referring pages were also good pages. The hyperlink vector voting algorithm (Li 1998) ranked web pages based on both the number of hyperlinks to it and the anchor-texts of the hyperlinks. Web usage mining (Srivastava et al. 2000) is the process of applying data mining techniques to discover web access patterns from a web log. Due to its greater relevance to this research, we review works on web usage mining in more detail.

The data used for web usage mining are web logs. A web log is a collection of data that explicitly records web surfers' behaviors of information searching in a website. Figure 3.2 shows a sample web log collected by a web server at the University of Arizona. Useful attributes for web usage mining in a web log include IP address, time and URL,

which explicitly describe who at what time accessed which web page. Additional attributes include status of a HTTP request and the count of bytes returned by a web server.

IP Address[3]	Time	Method/URL/Protocol	Status	Size
128.196.133 .16	(01/Sep/2001:05:3 8:33 -0700)	"GET /working/index.shtml HTTP/1.0"	200	7134
128.196.133 .16	(01/Sep/2001:05:3 8:34 -0700)	"GET /working/images/head-employ.gif HTTP/1.0"	200	765
128.196.133 .16	(01/Sep/2001:05:3 8:34 -0700)	"GET /working/images/work.jpg HTTP/1.0"	200	8864
128.196.133 .16	(01/Sep/2001:05:3 8:34 -0700)	"GET /working/images/staff-quicklinks1.gif HTTP/1.0"	200	1618

Figure 3.2: A Sample Web Log Collected By A Web Server At The University of Arizona

Projects of web usage mining are classified into two groups: general-purpose projects and specific-purpose projects (Srivastava et al. 2000). General-purpose projects, such as (Chen et al. 1996; Cooley et al. 1999a), focused on web usage mining in general. Cooley et al. proposed an architecture and specific steps for web usage mining and presented a method to identify potentially interesting patterns from mining results (e.g., patterns in which unlinked web pages are visited together frequently) (Cooley et al. 1999a). Chen et al. explored a new data mining capability to mine path traversal patterns from web logs (Chen el al. 1996). Specific-purpose projects focused on applications of web usage mining. Web usage mining can be used to improve organizations of websites. Adaptive

[3] To protect privacy of web users, IP addresses in this table are artificial IP addresses.

website project (Perkowitz and Etzioni 2000) used web visiting patterns learned from web logs to automatically improve organizations and presentations of websites. Spiliopoulou and Pohle exploited web usage mining to measure and improve the success of websites (Spiliopoulou and Pohle 2001). Lee and Podlaseck presented an interactive visualization system that provides users with abilities to actively interpret and explore web log data of online stores to evaluate the effectiveness of web merchandising (Lee and Podlaseck 2001). Web usage mining can also be used to personalize users' web surfing experience. In (Yan et al. 1996), clusters of visitors who exhibited similar information needs (e.g., visitors who accessed similar web pages) were discovered via web usage mining. These clusters could be used to classify new visitors and dynamically suggest hyperlinks for them. Mobasher et al. (Mobasher 2001) presented techniques to learn user preferences from web usage data using data mining techniques, such as association rule mining. Based on the learned preferences, dynamic hyperlinks could be recommended for active visiting sessions. Anderson et al. developed MINPATH, an algorithm that automatically suggests useful shortcut links in real time to improve wireless web navigations (Anderson et al. 2001). A complete survey of web usage mining research by year 2000 can be found in (Srivastava et al. 2000).

Research on browsing agents and recommendation systems is also related to this work. WebWatcher (Armstrong et al. 1995) and Letizia (Lieberman 1995) are two early examples of browsing agents. WebWatcher first asked a user's web visiting goal. It then predicted and highlighted hyperlinks that the user would click based on models learned from previous web visiting behaviors. Inspired by WebWatcher, personal WebWatcher

(Mladenic 1999) modeled web visiting interests for a specific web surfer based on her previously visited web pages. Another web browsing agent, Letizia learned a user's interests from the user's past behaviors. It explored the Web ahead of the user and used the interests learned to recommend web pages that the user would visit next. Content-based filtering and collaborative filtering are approaches to realizing recommendation systems. The content-based filtering approach has its root in information retrieval (Salton 1968). Recommendation systems using this approach, such as NewsWeeder (Lang 1995), recommended objects to a user based on the comparison between the contents of the objects and the user's profile. Recommendation systems using collaborative filtering approach, such as Ringo (Shardanand and Maes 995) and GroupLens (Resnick 1994), recommended objects to a user because other users with similar tastes to the given user liked these objects. The Fab system described in (Balabanovic and Shoham 1997) is based on the combination of the content-base filtering approach and the collaborative filtering approach.

Research employing only usage information, such as work described in (Chen el al. 1996), and research employing content information, such as work described in (Armstrong et al., 1995; Lang 1995), did not consider the information contained in the structure of a website. In (Cooley 1999a; Perkowitz and Etzioni 2000), the structure of a website was used to filter out uninteresting web visiting patterns (i.e., patterns in which directly linked web pages are visited together frequently). Although it is proper to exclude these uninteresting patterns in (Cooley 1999a; Perkowitz and Etzioni 2000),

these excluded uninteresting patterns provide valuable insights into hyperlink selection. We will discuss this in Section 3.4.1. The hyperlink selection approach proposed in this chapter is based on both web usage information and web structure information and considers both interesting and uninteresting web visiting patterns. Research in web structure mining that combines web content and web structure information is also related to this research. For example, HITS favors web pages (i.e., hub pages) that point to many other good pages (i.e., authority pages). LinkSelector also favors web pages that point to many other good pages. Unlike HITS, the goodness in LinkSelector is measured using web usage information (i.e., the frequency that the pages are visited together). Besides structurally related web pages, LinkSelector also considers structurally un-related web pages.

3.3 Problem Definition – Hyperlink Selection

It is desirable to design a portal page of a website (a) that can effectively facilitate web surfers to find information stored in the website (b) with limited number of hyperlinks placed in the portal page; and (c) that is frequently used/visited by web surfers when they search information in the website. We propose three metrics to measure the quality of a portal page – effectiveness, efficiency and usage, which address (a) – (c) mentioned above respectively. All three are calculated from web logs. A web log can be broken down into sessions with each session representing a sequence of consecutive web

accesses by the same visitor. For the convenience of readers, important notations used in this chapter are summarized in Table 3.1.

Table 3.1: Notation Summary

Notation	Description
s	the number of sessions in a web log
S_j	a session, for $j = 1,2,\cdots s$
H	the hyperlink pool of a website
$UHL(S_j)$	user-sought top-level web pages in S_j
l	the number of hyperlinks in a website
L_j	a hyperlink, for $j = 1,2,\cdots l$
P_{L_j}	the set of hyperlinks in the web page pointed to by L_j
PHL	the set of hyperlinks in a portal page
EHL	$EHL = \bigcup_{\forall L_j \in PHL} P_{L_j}$
HL	$HL = PHL \cup EHL$
N	the number of hyperlinks to be placed in a portal page
effectiveness(S_j)	$effectiveness(S_j) = \frac{\lvert UHL(S_j) \cap HL \rvert}{\lvert UHL(S_j) \rvert}$
effectiveness(log)	$effectiveness(\log) = \frac{\sum_{j=1}^{s} effectiveness(S_j)}{s}$
efficiency(S_j)	$efficiency(S_j) = \frac{\lvert UHL(S_j) \cap HL \rvert}{\lvert PHL \rvert}$
efficiency(log)	$efficiency(\log) = \frac{\sum_{j=1}^{s} efficiency(S_j)}{s}$
usage(log)	$usage(\log) = \sum_{j=1}^{s} \lvert UHL(S_j) \cap HL \rvert$

k-HS	a set of k hyperlinks L_i, where $L_i \in H$
σ(*k-HS*)	the support of a *k-HS*
SR	the set of structure relationships between hyperlinks in H
AR	the set of access relationships among hyperlinks in H
PRE_{L_i}	the preference of a hyperlink L_i, where $L_i \in H$
PHP	a set of pairs, in which, each pair consists of a hyperlink pair with group II relationship and its preference
PHS	a set of pairs, in which, each pair consists of a hyperlink set with group II relationship and its preference

The effectiveness of a portal page is measured as the degree of easiness to find user-sought top-level web pages[4] from the portal page. We denote s as the number of sessions in a web log, S_j as a session, for $j = 1,2,\cdots s$, and H as a hyperlink pool, which is a set of hyperlinks pointing to top-level web pages. In S_j, user-sought top-level web pages are web pages pointed to by hyperlinks in $UHL(S_j)$, which is the intersection of the hyperlinks clicked in S_j and H. Usually, web pages that are 1-2 clicks away[5] from a portal page can be easily found from the portal page. We denote l as the number of hyperlinks in a website, L_j as a hyperlink, for $j = 1,2,\cdots l$, P_{L_j} as the set of hyperlinks in

[4] We believe that a levelwise approach is appropriate for the design of a website. In this approach, the portal page is designed to find user-sought top-level web pages easily. Top-level web pages then are designed to locate user-sought web pages one level below easily. As the hyperlink selection approach proposed in this chapter can be applied to every subsequent level, consequently, websites designed in this approach will be easy to navigate to find user-sought information. In this chapter, we concentrate on designing the portal page to facilitate the search of top-level web pages.

[5] Web pages that are 1 click away from a portal page refer to web pages that are directly pointed to by hyperlinks in the portal page. Web pages that are 2 clicks away from a portal page are web pages that are pointed to by hyperlinks in web pages 1 click away.

the web page pointed to by L_j and *PHL* as the set of hyperlinks in the portal page of a website. *EHL* is the set of hyperlinks, where

$$EHL = \bigcup_{\forall L_j \in PHL} P_{L_j} \tag{3.1}$$

EHL consists of hyperlinks that are contained in web pages directly pointed to by the portal page. Web pages pointed to by hyperlinks in *HL* are 1-2 clicks away from the portal page and can be easily found from the portal page, where

$$HL = PHL \cup EHL \tag{3.2}$$

Example 3.1: As shown in Figure 3.3, the portal page of a web site contains hyperlinks L_1, L_2 and L_3. Web pages pointed to by hyperlinks L_1, L_2 and L_3 contain hyperlinks L_2 and L_4; hyperlink L_3; and hyperlinks L_5 and L_8 respectively.

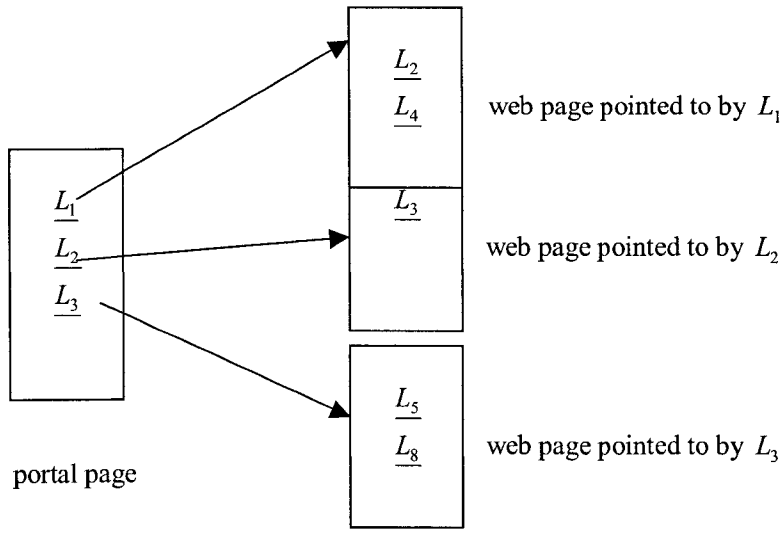


Figure 3.3: *PHL* and *EHL*

In this example, $PHL = \{L_1, L_2, L_3\}$, $P_{L_1} = \{L_2, L_4\}$, $P_{L_2} = \{L_3\}$ and $P_{L_3} = \{L_5, L_8\}$. According to (3.1), $EHL = \bigcup_{\forall L_j \in PHL} P_{L_j} = (P_{L_1} \cup P_{L_2} \cup P_{L_3}) = \{L_2, L_3, L_4, L_5, L_8\}$; according to (3.2), $HL = PHL \cup EHL = \{L_1, L_2, L_3, L_4, L_5, L_8\}$.

The effectiveness of a portal page can be measured in terms of the recall rate of the portal page at two different levels – session level and web log level. For a session S_j, the more hyperlinks in $UHL(S_j)$ found in *HL*, the more user-sought top-level web pages easily found from the portal page; hence, the higher the effectiveness of the portal page.

Definition 3.1: For a session S_j, the session level effectiveness of a portal page is defined as:

$$effectiveness(S_j) = \frac{|UHL(S_j) \cap HL|}{|UHL(S_j)|} \tag{3.3}$$

where $j = 1,2,\cdots s$ and $|X|$ denotes the cardinality of a set X. And the log level effectiveness of a portal page is defined as:

$$effectiveness(\log) = \frac{\sum_{j=1}^{s} effectiveness(S_j)}{s} \tag{3.4}$$

Example 3.2: In a session S_1, hyperlinks L_1, L_{10}, L_{11}, L_2, L_{13}, L_{14}, L_5, L_9, L_7 and L_{12} were clicked. Hyperlinks L_1, L_2, L_5 and L_7 are also elements of the hyperlink pool H. Hence, $UHL(S_1)$ is $\{L_1, L_2, L_5, L_7\}$. For the portal page given in Example 3.1, its session level effectiveness is,

$$effectiveness(S_1) = \frac{|UHL(S_1) \cap HL|}{|UHL(S_1)|} = \frac{|\{L_1, L_2, L_5\}|}{|\{L_1, L_2, L_5, L_7\}|} = \frac{3}{4} = 0.75$$

The result states that 75% of the user-sought top-level web pages in session S_1 can be easily found from the portal page.

Given the limited number of hyperlinks that can be placed in a portal page, it is desirable to have more user-sought top-level web pages easily found from the portal page (i.e., more hyperlinks in $UHL(S_j) \cap HL$) with fewer hyperlinks placed in the portal page (i.e., fewer hyperlinks in *PHL*). Motivated by this consideration, we define the efficiency of a portal page as the following.

Definition 3.2: For a session S_j, the session level efficiency of a portal page is defined as:

$$efficiency(S_j) = \frac{|UHL(S_j) \cap HL|}{|PHL|} \tag{3.5}$$

where $j = 1,2,\cdots s$ and $|X|$ denotes the cardinality of a set *X*. And the log level efficiency of a portal page is defined as:

$$efficiency(\log) = \frac{\sum_{j=1}^{s} efficiency(S_j)}{s} \tag{3.6}$$

Example 3.3: For the portal page given in Example 1 and the session S_1 given in Example 3.2, the session level efficiency of the portal page is,

$$efficiency(S_1) = \frac{|UHL(S_1) \cap HL|}{|PHL|} = \frac{|\{L_1, L_2, L_5\}|}{|\{L_1, L_2, L_3\}|} = \frac{3}{3} = 1$$

A well designed portal page should attract web surfers to use/visit it. Usage of a portal page measures how often a portal page is visited. As the portal page constructed by LinkSelector has not been used by web surfers, we measure its usage by counting the number of user-sought top-level web pages that can be easily found from the portal page (i.e., the number of hyperlinks in $UHL(S_j) \cap HL$). The measurement is an approximation of usage because ease of finding these web pages from the portal page will attract users to visit the portal page. We define usage measured at the web log level as below.

Definition 3.3: The log level usage of a portal page is defined as:

$$usage(\log) = \sum_{j=1}^{s} \left| UHL(S_j) \cap HL \right| \tag{3.7}$$

where $|X|$ denotes the cardinality of a set X.

Based on the three metrics presented above, we formally define the hyperlink selection problem as below.

Given: (1) the hyperlink pool H of a website;

(2) the number of hyperlinks to be placed in a portal page of the website – N, where

$$N < |H| .$$

Select N hyperlinks from the hyperlink pool H to include in a portal page.

Objective: maximize the effectiveness, efficiency and usage of the resulting portal page.

3.4 The LinkSelector Approach

As discussed in Section 3.1, it is computationally too expensive to find the optimal solution for the hyperlink selection problem. In this section, we present a heuristic solution method named LinkSelector. Compared with the current practice of hyperlink selection based on domain experts' experience, our method incorporates patterns extracted from the structure of a website and those extracted from a web log, which records web surfers' behaviors of information searching in the website. Hence, LinkSelector is more objective and reflects web surfers' perspectives on which hyperlinks should be selected while current practice of hyperlink selection is subjective and reflects only domain experts' perspectives. We introduce LinkSelector in Section 3.4.1. A detailed description of LinkSelector is illustrated step by step in Section 3.4.2 through Section 3.4.6. We discuss time complexity of LinkSelector in Section 3.4.7.

3.4.1 Overview of LinkSelector

Hyperlinks in a hyperlink pool may be directly connected to each other (i.e., one hyperlink is contained in a web page pointed to by another hyperlink) or accessed together in a session. Accordingly, we categorize relationships among hyperlinks in a hyperlink pool into two types – structure relationship and access relationship. For any pair of hyperlinks L_i and L_j in H, L_i has a structure relationship with L_j , denoted as $L_i \rightarrow L_j$, if and only if $L_j \in P_{L_i}$ (i.e., L_j is a hyperlink on the page pointed to by L_i). L_i is the initial hyperlink and L_j is the terminal hyperlink in this structure relationship.

Example 3.4: As shown in Figure 3.4, web page 1 contains hyperlinks L_1 and L_3; web page 2 , which is pointed to by hyperlink L_1, contains hyperlinks L_2, L_4, L_6 and L_8 ; and web page 3 , which is pointed to by hyperlink L_3, contains hyperlink L_5 and L_7. All hyperlinks are elements of the hyperlink pool *H*.

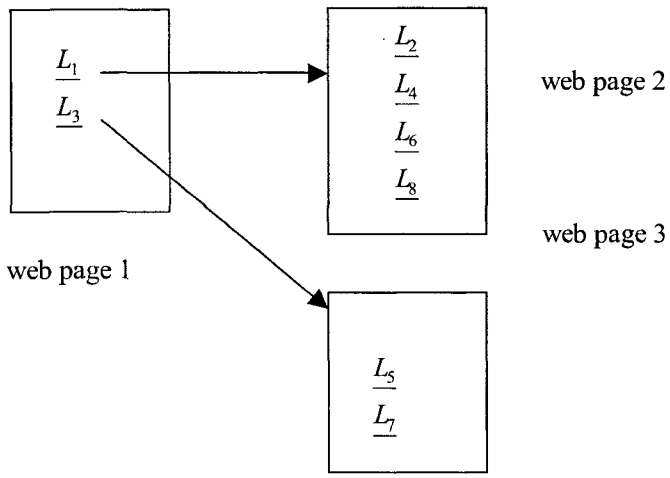


Figure 3.4: Structure Relationship

In this example, $P_{L_1} = \{L_2, L_4, L_6, L_8\}$ and $P_{L_3} = \{L_5, L_7\}$. Structure relationship $L_1 \rightarrow L_2$ exists, in which L_1 is the initial hyperlink and L_2 is the terminal hyperlink. Similarly, structure relationships $L_1 \rightarrow L_4$, $L_1 \rightarrow L_6$, $L_1 \rightarrow L_8$, $L_3 \rightarrow L_5$ and $L_3 \rightarrow L_7$ also hold.

For a hyperlink acts as the initial hyperlink in *M* structure relationships, placing it in a portal page makes it an element of *PHL* and the *M* terminal hyperlinks in all the structure relationships elements of *EHL*. As discussed in Section 3.3, web pages pointed to by

hyperlinks in *HL* (i.e., $PHL \cup EHL$) can be easily found from a portal page. Therefore, the more structure relationships a hyperlink participates in as the initial hyperlink, the more top-level web pages can be easily found from a portal page if the hyperlink is placed in the portal page. In Example 3.4, L_1 participates in four structure relationships (i.e., $L_1 \rightarrow L_2$, $L_1 \rightarrow L_4$, $L_1 \rightarrow L_6$ and $L_1 \rightarrow L_8$) as the initial hyperlink. Hence, placing L_1 in a portal page enables five top-level web pages (i.e., pages pointed to by L_1, L_2, L_4, L_6 and L_8) to be easily found from the portal page.

We use access relationships to reflect the patterns that some hyperlinks are visited together in a session. In a web log, session is identified using IP address and a time-out (see Section 3.4.3.1). The LinkSelector approach is based on both access relationships and structure relationships. Before introducing access relationships, we introduce hyperlink sets first. A hyperlink set *k-HS* is a set of *k* hyperlinks L_i, where $L_i \in H$. For a web log, the support of a *k-HS* (denoted as $\sigma(k\text{-}HS)$) is the ratio of the number of sessions in which the *k-HS* is accessed over the total number of sessions in the web log. For a *k-HS*, where $k \geq 2$, there exists an access relationship among elements in the *k-HS* if and only if its support $\sigma(k\text{-}HS)$ is greater than a pre-defined threshold. $\sigma(k\text{-}HS)$ is called the support of the access relationship.

For hyperlinks in a hyperlink pool, their pairwise relationships can be categorized into four groups as shown in Figure 3.5.

Structure Relationship / Access Relationship	YES	NO
YES	I	II
NO	III	IV

Figure 3.5: Pairwise Relationships for Hyperlinks In a Hyperlink Pool

- Group I relationship indicates that both structure relationship and access relationship hold between two hyperlinks. As we have discussed, hyperlinks participating in more structure relationships as initial hyperlinks will be selected for the portal page over other hyperlinks with respect to increasing the number of hyperlinks in *HL*. For a structure relationship $L_i \rightarrow L_j$, the support of the access relationship between L_i and L_j -- $\sigma(\{L_i, L_j\})$, reflects the quality of the structure relationship. The higher the support $\sigma(\{L_i, L_j\})$, the higher the possibility that L_i and L_j will be accessed together in future visits[6]. Hence, hyperlinks participating in higher quality structure relationships as initial hyperlinks will be selected for the portal page over other hyperlinks with respect to increasing the quality (i.e., possibility to be visited) of hyperlinks in *HL*. Increasing the number and the quality of hyperlinks in *HL* could increase the hits of user-sought top-level web pages in *HL* (i.e., $|UHL(S_j) \cap HL|$), which in turn could increase the effectiveness, efficiency and usage of a portal page, according to (3.3), (3.5) and (3.7). In this regard, group I relationship which was

[6] This is based on an assumption that web visiting patterns are coherent in past and future visits (Perkowitz and Etzioni 2000).

regarded as uninteresting in previous research (Perkowitz and Etzioni 2000; Cooley 1999a), provides us with two indicators of hyperlink preference in hyperlink selection: the number of structure relationships a hyperlink participates in as the initial hyperlink and the qualities of these structure relationships measured as supports of access relationships between the hyperlink and its terminal hyperlinks. We describe how to calculate the preference of a hyperlink using these two indicators in Section 3.4.4.

- Group II relationship indicates that an access relationship but not a structure relationship exists between two hyperlinks. As hyperlinks with group II relationship are structurally unlinked, in order to navigate from the web page pointed to by one hyperlink to the web page pointed to by the other, web visitors have to explore the website to find the path. This creates inconvenience for web surfing and the situation becomes worse as these two hyperlinks are accessed together frequently (i.e., access relationship between the hyperlinks). In contrast, if these two hyperlinks were placed together in a portal page, they could easily be found from the portal page, which could increase the hits of user-sought top-level web pages in HL (i.e., $|UHL(S_j) \cap HL|$) and the effectiveness, efficiency and usage of a portal page (i.e., according to (3.3), (3.5) and (3.7)). Hence, hyperlink pairs with group II relationship are preferred to hyperlink pairs without in hyperlink selection. For a hyperlink pair with group II relationship, the support of the access relationship between its hyperlinks is the determining factor for its preference over other hyperlink pairs with

group II relationship in hyperlink selection. We describe how to calculate the preference of a hyperlink pair for group II relationship in Section 3.4.5.

- Group III relationship indicates that a structure relationship but not an access relationship exists between two hyperlinks. This relationship reveals that the web page pointed to by the initial hyperlink in a structure relationship contains a rarely visited hyperlink which is the terminal hyperlink in the structure relationship. Hence, group III relationship reveals a design problem with internal pages. As hyperlink selection focuses on choosing hyperlinks for a portal page, we do not discuss group III relationship in this chapter.

- Group IV relationship does not reveal any patterns between hyperlinks; thus, is not considered in this research.

LinkSelector employs group I and group II relationships to calculate preferences of hyperlinks and preferences of hyperlink pairs respectively. Based on preferences calculated, an algorithm is developed to extract N hyperlinks from the given hyperlink pool. We outline LinkSelector in Figure 3.6. Steps in this algorithm are described in Section 3.4.2 through Section 3.4.6.

Input: H : the hyperlink pool of a website
N : the number of hyperlinks to be placed in the portal page of the website
Output: *PHL*: the set of hyperlinks in the portal page of the website

Discover structure relationships;
Discover access relationships;
Calculate preferences of hyperlinks based on group I relationship;
Calculate preferences of hyperlink pairs based on group II relationship;
Select hyperlinks.

Figure 3.6: The Sketch of LinkSelector

3.4.2 Discover Structure Relationships

LinkSelector discovers structure relationships between hyperlinks in a hyperlink pool by parsing the web pages the hyperlinks point to. The web page pointed to by each hyperlink L_i in H is retrieved and parsed. A structure relationship $L_i \rightarrow L_j$ is added to structure relationships *SR* if L_j appears in the web page L_i points to, L_j and L_i are different and L_j is an element of H.

3.4.3 Discover Access Relationships

3.4.3.1 Web Log Preprocessing

Access relationships among hyperlinks can be extracted from a web log. A raw web log collected from a web server needs to be preprocessed before meaningful access

relationships can be extracted (Cooley et al. 1999b). In this research, we divide the preprocessing task into two steps – web log cleaning and session identification.

In web log cleaning, two types of web log records are removed. First, web log records with the value of the status attribute greater than 400 are deleted because they record failed web access. Second, web log records recording accessory requests to a web page request, such as picture requests, are also removed. As a raw web log records every file request sent to a web server, one web access could result in several web log records. For example, an access to a web page with two pictures will result in three web log records, one for the web page and the other two for the pictures. Web log records recording accesses to web pages are sufficient to describe web surfers' information searching behaviors.

The basic processing unit for extracting access relationships is a session. A web log needs to be divided into sessions before the extraction of access relationships. However, in HTTP protocol, as connections between web clients and a web server are stateless, there is no notion of session in a web log. One method of dividing a web log into sessions is based on timeout. If the time between page requests from the same user exceeds a certain limit, it is assumed that the user has started another session (Cooley et al. 1999b). Some commercial tools use 30 minutes as a default timeout. Catledge and Pitkow calculated a timeout of 25.5 minutes based on empirical data (Catledge and Pitkow 1995). Another method is to modify a web server to encode session identifiers in web pages transferred between clients and a web server (Yan et al. 1996). As the second method requires

modification of a web server, it is not convenient and practical for many websites. We adopt the timeout method to identify sessions. Web log records are first sorted by client IP address then by access time. We use a 30-minute timeout to divide web log records generated by the same IP address into sessions.

3.4.3.2 Mine Access Relationships From The Preprocessed Web Log

Once a raw web log has been cleaned and divided into sessions, large itemsets discovery algorithms proposed in association rule mining can be applied to extract access relationships from it. Association rule mining (Agrawal and Srikant 1994) is defined on a set of items $L = \{i_1, i_2, \cdots, i_k\}$. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq L$. The support of an itemset (i.e., a set of items) in D is the fraction of all transactions containing the itemset. An itemset is called large if its support is greater than a user-specified support threshold. The most important step in association rule mining is to find large itemsets and their supports. Various algorithms for finding large itemsets, such as Apriori (Agrawal and Srikant 1994), are in use.

In the case of mining access relationships, sessions correspond to transactions, hyperlinks correspond to items and hyperlink sets correspond to itemsets. Applying the Apriori algorithm on the preprocessed web log, all hyperlink sets that have access relationships among their elements can be found and their corresponding supports can be calculated. The Apriori algorithm can be found in (Agrawal and Srikant 1994) and is skipped in this

chapter. The output of mining access relationships is a set of pairs denoted as *AR*, in which, each pair contains a hyperlink set and its corresponding support.

3.4.4 Calculate Preferences of Hyperlinks

As discussed in Section 3.4.1, with respect to increasing the effectiveness, efficiency and usage of a portal page, the preference of a hyperlink in hyperlink selection is determined by two factors: the number of structure relationships it participates in as the initial hyperlink and the qualities of these structure relationships measured as supports of access relationships between it and its terminal hyperlinks. Based on these two factors, we define the preference of a hyperlink as below.

Definition 3.4: For a hyperlink L_i,where $L_i \in H$, and L_i participates in M structure relationships as the initial hyperlink, $(L_i \rightarrow L_{j_m}) \in SR$, for $m = 1,2,\cdots M$, the preference of the hyperlink L_i is defined as:

$$PRE_{L_i} = \sum_{m=1}^{M} (\sigma(\{L_i, L_{j_m}\}) * coeff) \tag{3.8}$$

where $coeff = \begin{cases} 1 & \text{if } (\{L_i, L_{j_m}\}, \sigma(\{L_i, L_{j_m}\})) \in AR \\ 0 & \text{otherwise} \end{cases}$

Example 3.5: Given a hyperlink pool $H = \{L_1, L_2, L_3, L_4, L_5, L_6, L_7\}$, structure relationships *SR*, the minimum threshold for access relationship – 0.002, and access relationships *AR*,

$$SR = \{L_1 \rightarrow L_2, L_1 \rightarrow L_3, L_1 \rightarrow L_4, L_2 \rightarrow L_5, L_3 \rightarrow L_4, L_3 \rightarrow L_6, L_4 \rightarrow L_5, L_5 \rightarrow L_6,$$

$$L_5 \rightarrow L_7, L_6 \rightarrow L_3, L_7 \rightarrow L_3, L_7 \rightarrow L_4\}$$

$$AR = \{ (\{L_1, L_2\}, 0.022), (\{L_1, L_3\}, 0.018), (\{L_1, L_4\}, 0.01), (\{L_1, L_5\}, 0.002), (\{L_1, L_6\}, 0.014),$$

$$(\{L_1, L_7\}, 0.014), (\{L_2, L_5\}, 0.007), (\{L_2, L_6\}, 0.006), (\{L_2, L_7\}, 0.008), (\{L_3, L_6\}, 0.008),$$

$$(\{L_3, L_7\}, 0.012), (\{L_4, L_5\}, 0.01), (\{L_5, L_6\}, 0.03), (\{L_6, L_7\}, 0.005), (\{L_1, L_6, L_7\}, 0.003)\}$$

Hyperlink L_1 participates in three structure relationships as the initial hyperlink. According to (3.8),

$$PRE_{L_1} = \sum_{m=1}^{3} (\sigma(\{L_1, L_{j_m}\} * coeff) = \sigma(\{L_1, L_2\}) * 1 + \sigma(\{L_1, L_3\}) * 1 + \sigma(\{L_1, L_4\}) * 1$$

$$= 0.022 + 0.018 + 0.01 = 0.05$$

Similarly, we get, $PRE_{L_2} = 0.007, PRE_{L_3} = 0.008, PRE_{L_4} = 0.01, PRE_{L_5} = 0.03, PRE_{L_6} = 0.008$

and $PRE_{L_7} = 0.012$.

3.4.5 Calculate Preferences of Hyperlink Pairs

As discussed in Section 3.4.1, with respect to increasing the effectiveness, efficiency and usage of a portal page, hyperlink pairs with group II relationship are preferred to

hyperlink pairs without in hyperlink selection. For a hyperlink pair with group II relationship, the support of the access relationship between its hyperlinks is the determining factor for its preference over other hyperlink pairs with group II relationship in hyperlink selection. We set preferences of hyperlink pairs without group II relationship to be 0 and those of hyperlink pairs with group II relationship to be the supports of the access relationships between its hyperlinks.

Both hyperlink pairs with group II relationship and their preferences can be extracted from access relationships *AR*. For a 2-*HS* (i.e., a hyperlink set with two hyperlinks as elements) in *AR*, if there exists no structure relationship between its elements, then the 2-*HS* is a hyperlink pair with group II relationship and its support $\sigma(2\text{-}HS)$ is the preference of the hyperlink pair. We denote *PHP* as a set of pairs, in which, each pair consists of a hyperlink pair with group II relationship and the preference of the hyperlink pair.

Example 3.6: For a 2-*HS* $\{L_1, L_5\}$ in access relationships *AR* given in Example 3.5, since there is no structure relationship between hyperlinks L_1 and L_5, $\{L_1, L_5\}$and its support become an element of *PHP* shown below. Preferences of hyperlink pairs outside *PHP* are set to be 0.

$$PHP = \{ (\{L_1, L_5\}, 0.002), (\{L_1, L_6\}, 0.014), (\{L_1, L_7\}, 0.014), (\{L_2, L_6\}, 0.006), (\{L_2, L_7\}, 0.008),$$

$$(\{L_6, L_7\}, 0.005)\}$$

LinkSelector considers not only hyperlink sets with 2 elements (i.e., hyperlink pairs) that have group II relationship but also hyperlink sets with more than 2 elements that have group II relationship. Discussions on hyperlink pairs with group II relationship can be extended to include hyperlink sets with more than 2 elements.

Definition 3.5: A hyperlink set *k-HS*, where $k \geq 2$, is a hyperlink set with group II relationship if and only if,

- there exists an access relationship among its hyperlinks, and
- there is no structure relationship between any pair of its hyperlinks.

Similarly, we set preferences of hyperlink sets without group II relationship to be 0. And, the preference of a hyperlink set with group II relationship is set to be the support of the access relationship among its hyperlinks. We denote *PHS* as a set of pairs, in which, each pair consists of a hyperlink set with group II relationship and the preference of the hyperlink set.

Example 3.7:*PHS* extracted from access relationships *AR* in Example 3.5 is,

$$PHS = \{ (\{L_1, L_5\}, 0.002), (\{L_1, L_6\}, 0.014), (\{L_1, L_7\}, 0.014), (\{L_2, L_6\}, 0.006), (\{L_2, L_7\}, 0.008),$$

$$(\{L_6, L_7\}, 0.005), (\{L_1, L_6, L_7\}, 0.003)\}$$

According to Definition 3.5, hyperlink set $\{L_1, L_6, L_7\}$ is added to *PHS*. Preferences of hyperlink sets outside *PHS* are set to be 0.

The algorithm to extract *PHS* from access relationships *AR* is straightforward and skipped in this chapter.

3.4.6 Select Hyperlinks

In the preceding sections, we have defined preferences of hyperlinks and preferences of hyperlink sets, that reflect the contribution of the selected hyperlinks to the three metrics of a portal page. In this section, we define an approximation of the hyperlink selection problem using the preferences as the following.

Given: (1) the hyperlink pool H of a website;

(2) the number of hyperlinks to be placed in a portal page of the website – N, where

$$N < |H|.$$

Select N hyperlinks from the hyperlink pool H to construct *PHL* (i.e., the set of hyperlinks in a portal page)

Objective: maximize the following objective function,

$$SCORE(PHL) = \sum_{L \in PHL} PRE_L + \sum_{k-HS \subseteq PHL} (\sigma(k\text{-HS})) \tag{3.9}$$

where $(k\text{-HS}, \sigma(k\text{-HS})) \in PHS$ and $2 \le k \le |PHL|$. $SCORE(PHL)$ which is the score of PHL, includes two parts: the preferences of hyperlinks in PHL and the preferences of hyperlink sets in PHL.

We propose an algorithm, namely the greedy aggregation algorithm, to address the above problem heuristically. The algorithm is based on an average-score matrix whose indexes are hyperlink sets and elements are average scores of the merged indexes (i.e., hyperlink sets), which are defined below.

Definition 3.6: For any two hyperlink sets C_i and C_j, the average score of $C_i \cup C_j$ (i.e., the merge of C_i and C_j), $AVG(C_i \cup C_j)$ is,

$$AVG(C_i \cup C_j) = \frac{\sum_{L \in |C_i \cup C_j|} PRE_L + \sum_{k\text{-}HS \subseteq |C_i \cup C_j|} (\sigma(k\text{-HS}))}{|C_i \cup C_j|} \tag{3.10}$$

where $(k\text{-HS}, \sigma(k\text{-HS})) \in PHS$ and $2 \le k \le |C_i \cup C_j|$.

Example 3.8: Given preferences of hyperlinks and hyperlink sets calculated in Example 3.5 and Example 3.7, for two hyperlink sets $C_1 = \{L_2, L_3\}$ and $C_2 = \{L_7\}$, according to (3.10),

$$AVG(C_1 \cup C_2) = \frac{PRE_{L_2} + PRE_{L_3} + PRE_{L_7} + \sigma(\{L_2, L_7\})}{3} = 0.0117$$

$AVG(C_i \cup C_j)$ (see (3.10)) is designed based on the following two considerations: (a) its nominator is used to reflect the objective function (see (3.9)); and (b) its denominator is introduced to avoid the possibility that the number of hyperlinks in hyperlink sets could impact the value of $AVG(C_i \cup C_j)$. Using the greedy aggregation algorithm, each hyperlink in H is initially placed in a unique hyperlink set and the initial average-score matrix is constructed based on (3.10). The algorithm merges the two hyperlink sets having the highest average score. After merging the two hyperlink sets, the average-score matrix is updated by re-computing average scores related to the merged set. This process is repeated until the number of hyperlinks in a hyperlink set is greater than or equal to N[7]. This hyperlink set is output as PHL[8]. We summarize the greedy aggregation algorithm in Figure 3.7.

[7] The proposed greedy algorithm is practical but generates a local optimal solution.
[8] If the number of hyperlinks in the result hyperlink set is larger than N, then N hyperlinks are selected from it according to their individual preferences, from high to low, to construct PHL .

Input: H: the hyperlink pool of a website

PRE_L : preferences of hyperlinks, where $L \in H$

PHS : hyperlink sets and their preferences

N: the number of hyperlinks to be placed in a portal page

Output: PHL : the set of hyperlinks in the result portal page

distribute each hyperlink in H into a unique hyperlink set;
max_size = 1; */*max_size: maximum number of hyperlinks in hyperlink sets*/*
initialize AVG; */*AVG: the average-score matrix*/*
While (max_size < N)
    merge hyperlink sets with the highest average score into C; */*C: merged set*/*
    If (size(C) > max_size) */*size(C): number of hyperlinks in C*/*
        max_size = size(C);
    End if
    update AVG ; */*only need to re-compute average scores related to the merged set*/*
End while
$PHL = C$;

Figure 3.7: The Greedy Aggregation Algorithm

3.4.7 Time Complexity of LinkSelector

To compute time complexity of LinkSelector, time complexity for each step in the algorithm is considered first. We denote the number of hyperlinks in the given hyperlink pool H as n_l, the number of sessions in a web log as n_s, the number of elements in access relationships AR as n_{AR} and the number of elements in structure relationships SR as n_{SR}. Structure relationships are discovered in step 1. In this step, pages pointed to by all hyperlinks in H are retrieved and parsed to extract structure relationships. If the

average number of hyperlinks in all retrieved pages is β, time complexity for step 1 is $O(n_l \times \beta)$. Usually, β is much less than n_l. Hence, time complexity for step 1 is $O(n_l)$. We employ the Apriori algorithm to discover access relationships AR[9] in step 2. Apriori goes through all sessions in a web log α rounds to discover access relationships AR. Therefore, time complexity for step 2 is $O(n_s \times \alpha)$. As α is much less than n_s (e.g., in our experiment discussed in Section 3.5, α equals to 6 while n_s is 262K), time complexity for step 2 is the order of $O(n_s)$. Preferences of hyperlinks are computed in step 3. If the average number of structure relationships a hyperlink participates in as the initial hyperlink is γ, time complexity for step 3 is $O(n_l \times \gamma)$. Usually, γ is much less than n_l. Hence, time complexity for step 3 is $O(n_l)$. In step 4, each hyperlink set in access relationships AR is checked to see whether there is structure relationship between its elements. Hence, time complexity for step 4 $O(n_{AR})$. One $n_l \times n_l$ average-score matrix is created in step 5, which needs $O(n_l^2)$ time. At worst, it also needs $O(n_l^2)$ time to re-compute the similarity matrix. Hence, time complexity for step 5 is the order of $O(n_l^2)$. Combining time complexities of all 5 steps, we get the time complexity of LinkSelector as $O(n_l^2 + n_s + n_{AR})$.

[9] We assume that web log is cleaned and sessions are identified before applying LinkSelector.

3.5 Experiment Results

To evaluate the performance of LinkSelctor, we compare it with other commonly used hyperlink selection approaches. Section 3.5.1 describes the experimental data while Section 3.5.2 presents and analyzes the comparison results.

3.5.1 Experimental Data

We performed the experiments on the University of Arizona website because it is large enough and generates sufficient web logs to permit comparisons of different hyperlink selection approaches. Three months' web logs – September 2001, February 2002 and July 2002, collected from the university web server were used separately in the experiments. The three months' web logs can be considered as representative web logs for three major time periods at the university – Fall semester (Sep. 2001), Spring semester (Feb. 2002) and Summer break (July 2002) respectively. As visitors to the university website changes from period to period, visitors' web access patterns could also change. Furthermore, the university website was re-designed and went live in July, 2002. The three months' web logs and web structure information thus provide an ideal testbase to test LinkSelector under different situations.

The raw web log of Sep. 2001 contained 10 million records and the cleaned web log had 4.2 million records, sufficiently rich for extracting web visiting patterns. We used a 30-minute timeout to divide the cleaned web log into 344K sessions. 23 days of the cleaned web log were used as the training data for different hyperlink selection approaches, which

contained 262K sessions; the last 7 days of the cleaned web log were used as the testing data, which contained 82K sessions. Table 3.2 summarizes the statistics of the three months' web logs. By applying the Apriori algorithm to the three months' training data, we derived 184, 209 and 167 access relationships, respectively, for the training data in Sep. 2001, Feb. 2002 and July 2002. Access relationships shared by all three training data totaled 136[10]. For example, the access to /visiting/alumni.shtml was much more active in Feb. 2002, compared with the other two months. Specially, the most active access relationship involving /visiting/alumni.shtml in Feb. 2002 was between /visiting/alumni.shtml and /spotlight/index.shtml, which points to campus news. A possible reason could be that several major alumni gatherings were held in Feb. 2002.

Table 3.2: Summary of the Experimental Data

Time Span	Total Data	Training Data	Testing Data	Number of Access Relations	Number of Structure Relationships
Sep. 2001	344K sessions	262K sessions (23 days)	82K sessions (7 days)	184	594
Feb. 2002	282K sessions	208K sessions (21 days)	74K sessions (7 days)	209	594
July 2002	323K sessions	245K sessions (24 days)	78K sessions (7 days)	167	578

[10] Two access relationships are regarded the same if, (1) hyperlinks in the two access relationships are same; and (2) the deviation between the supports of the two access relationships is no more than 10%.

The hyperlink pool used in all three time periods consists of 743 hyperlinks in the index page of the university website (http://www.arizona. edu/index/webindex.shtml). Among these, 110 hyperlinks pointed to web pages at the university web server and the remaining 633 hyperlinks pointed to web pages at other web servers. Because the web log was collected from the university web server, access relationships among the 633 hyperlinks pointing to web pages at other web servers could not be mined from the collected web log. Therefore, the hyperlink pool used in our experiments consists of the 110 hyperlinks pointing to web pages at the university web server. From these, 594 structure relationships were extracted before the website redesign; 578 structure relationships were extracted after the website redesign. Between the structure relationships extracted before and after the website redesign, 533 structure relationships were the same. Some notable structure relationship changes after the redesign included: structure relationships between several hyperlinks and hyperlink /shared/aboutua.shtml, such as structure relationship "/shared/athletics.shtml → /shared/aboutua.shtml", emerged. After the website redesign, some hyperlinks were renamed (e.g., /shared/sports-entertain.shtml was renamed /home/athletics.shtml). We use old names throughout the chapter to avoid confusion.

3.5.2 Performance Comparison With Expert Selection And Top-Link Selection

Current practices of hyperlink selection, namely expert selection, rely on domain experts' (e.g., webmasters) experiences. Hyperlinks contained in the current portal page (http://www.arizona.edu) of the university web site are hyperlinks selected by domain

experts. Another simple approach to hyperlink selection, namely top-link selection, selects hyperlinks with top N access frequencies from the given hyperlink pool, where N is the number of hyperlinks to be placed in a portal page. Table 3.3 lists 6 hyperlinks selected by LinkSelector, domain experts[11] and top-link selection using the Sep. 2001 training data. Figure 3.8 illustrates the performance comparison among the three approaches using the Sep. 2001 testing data.

Table 3.3. Hyperlinks Selected by LinkSelector, Domain Experts and Top-link Selection (N=6; Sep. 2001)

No.	LinkSelector	Expert Selection	Top-Link Selection
1	/index/alldepts-index.shtml	/student_link	/student_link
2	/shared/sports-entertain.shtml	/index/alldepts-index.shtml	/index/alldepts-index.shtml
3	/working/teaching.shtml	/newschedule/parse-schedule-new.cgi	/newschedule/parse-schedule-new.cgi
4	/shared/aboutua.shtml	/phonebook	/phonebook
5	/shared/getting-around.shtml	/shared/sports-entertain.shtml	/shared/sports-entertain.shtml
6	/spotlight/index.shtml	/shared/libraries.shtml	/shared/athletics.shtml

[11] In this experiment, k hyperlinks selected by domain experts are k hyperlinks chosen from the portal page designed by domain experts and these chosen hyperlinks are top-k frequently accessed hyperlinks among all hyperlinks in the portal page.

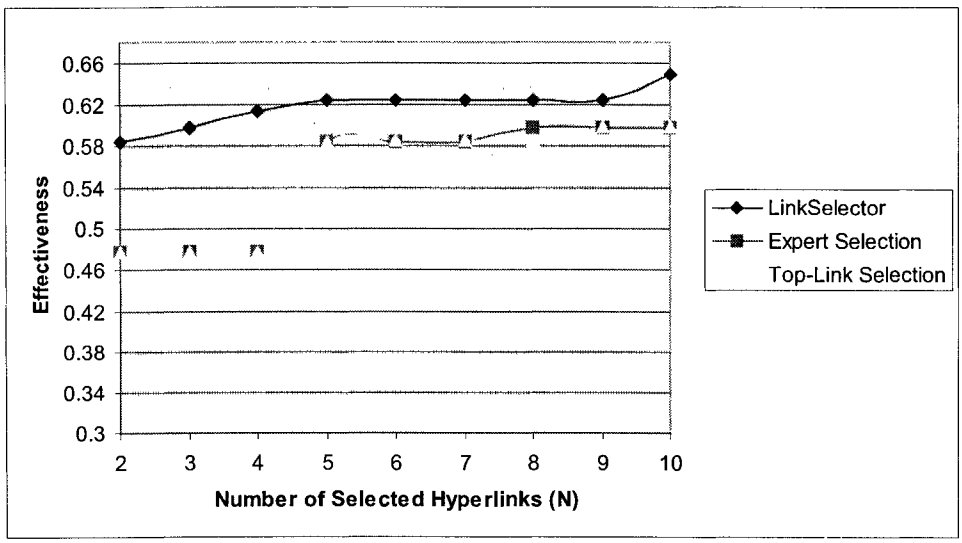


Figure 3.8: (a) Effectiveness Comparison Among LinkSelector, Expert Selection and Top-link Selection (Sep. 2001)

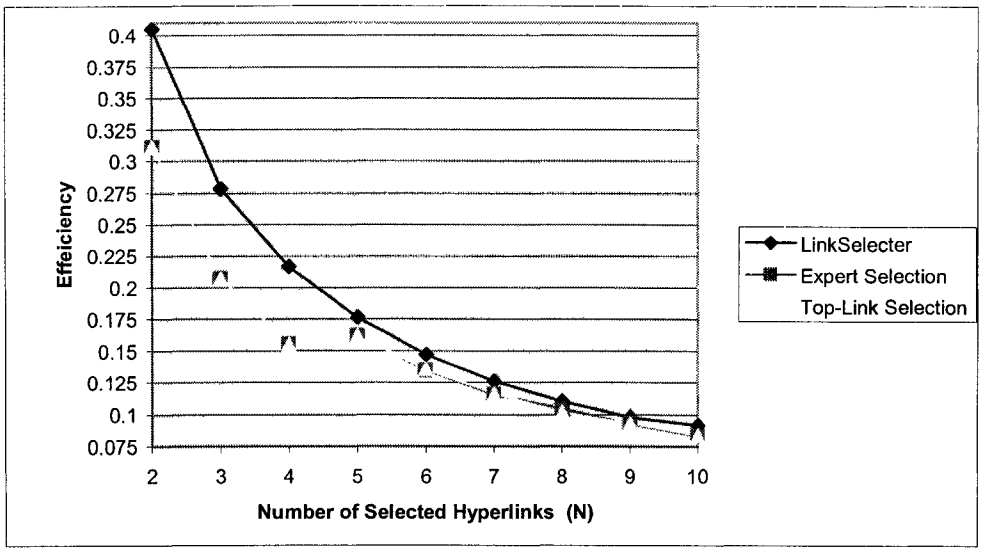


Figure 3.8: (b) Efficiency Comparison Among LinkSelector, Expert Selection and Top-link Selection (Sep. 2001)

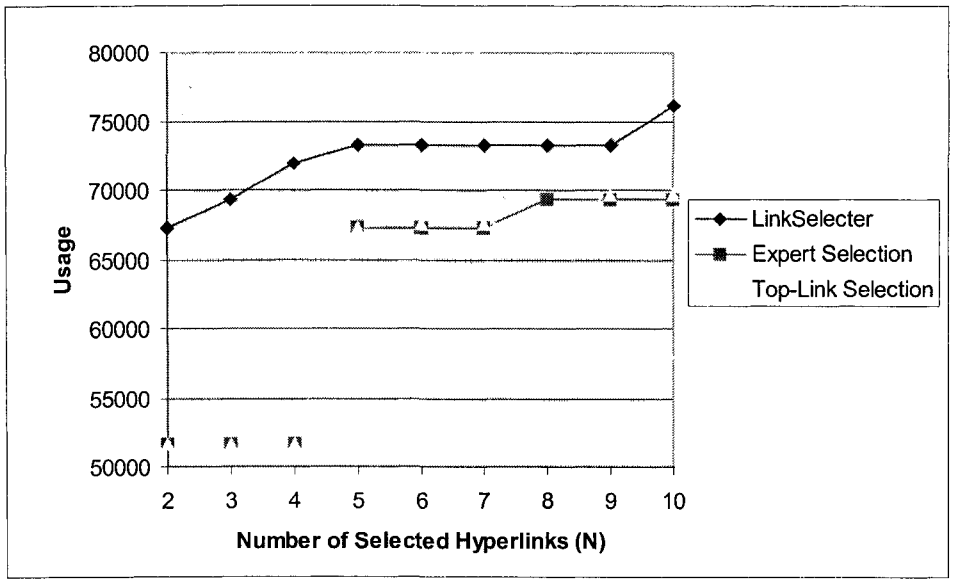


Figure 3.8: (c) Usage Comparison Among LinkSelector, Expert Selection and Top-link Selection (Sep. 2001)

On average, LinkSelector outperformed both expert selection and top-link selection with a 12.7% increase in the effectiveness. Given the large number of visiting sessions (e.g., 11.5k sessions per day at the website of the University of Arizona), this was a big improvement in ease of finding user-sought top level web pages. The improvement decreased from 22.1% to 8.4% as the selection ratio (i.e., the ratio of the number of the selected hyperlinks over the total number of hyperlinks in a hyperlink pool) increased from 1.8% (i.e., 2/110) to 9.1% (i.e., 10/110). However, even at the selection ratio of 9.1%, which is more than double the selection ratio for the portal page of the University of Arizona website (i.e., 32/743=4.3%), the improvement in effectiveness (i.e., 8.4%) was

still apparent. Moreover, at high selection ratio, such as 8.2% (i.e., $N = 9$), 4238[12] more user-sought top level web pages could be easily accessed from the portal page constructed using LinkSelector than from that constructed utilizing expert selection. As the testing data covers a 7-day time period, the average saving per day was 605. Compared with expert selection and top-link selection, LinkSelector improved the efficiency by 16.9% on average. Similarly, the improvement for the efficiency decreased from 30.2% to 9.3% as the selection ratio increased. Compared with expert selection and top-link selection, LinkSelector also improved the usage by an average of 17.0%. The improvement decreased from 30.2% to 9.4% as the selection ratio was increased.

The improvements were attributed to the relationships among hyperlinks considered in LinkSelector but missed in the other two approaches. For example, hyperlinks /shared/sports-entertain.shtml and /shared/athletics.shtml were on a popular path to web pages on sports and entertainments. The first hyperlink was the starting point of the path and the second one was the second link on this path. Therefore, both of them had high access frequency and were selected by top-link selection. However, top-link selection failed to consider that there was a structure relationship between these two hyperlinks. In this structure relationship, hyperlink /shared/sports-entertain.shtml was the initial hyperlink and hyperlink /shared/athletics.shtml was the terminal hyperlink. It was natural to navigate from /shared/sports-entertain.shtml to /shared/athletics.shtml and find web pages on sports. Hence, it was unnecessary to put both of them in a portal page. Applying

---

[12] The number was derived by comparing $\sum_{j=1}^{s} \left| UHL(S_j) \cap HL \right|$ between LinkSelector and expert selection.

group I relationship, LinkSelector selected only the starting point of the path -- /shared/sports-entertain.shtml. Both top-link selection and expert selection failed to consider group II relationships among hyperlinks (i.e., hyperlinks that are structurally unrelated but access related). For example, hyperlinks /shared/sports-entertain.shtml and /shared/aboutua.shtml were structurally unrelated hyperlinks. However, a large number of sessions looking for information regarding sports and entertainment at the university (i.e., /shared/sports-entertain.shtml) also tried to learn something about the university (i.e., /shared/aboutua.shtml). Placing both hyperlinks in a portal page could save web surfers' efforts of finding the path from one topic to the other. Applying group II relationship, LinkSelector selected both hyperlinks.

As described in Section 3.5.1, there are changes regarding web visiting patterns and the website structure in Feb. and July 2002 compared with Sep. 2001. However, neither expert selection nor top-link selection were sensitive to these changes. Before and after the website redesign, the majority of the hyperlinks in the homepage of the university website were same. Although there were more visits to hyperlink /visiting/alumni.shtml in Feb. 2002, top-link selection did not select it because it was not one of the top-visited hyperlinks. LinkSelector, on the other hand, was sensitive to these changes. Table 3.4 lists hyperlinks selected by LinkSelector in the three time periods. As visits to /visiting/alumni.shtml were much more active in Feb. 2002, especially, there existed an access relationship between /visiting/alumni.shtml and /spotlight/index.shtml and no structure relationship between them, LinkSelector picked /visiting/alumni.shtml by

considering group II relationship. In July 2002, as the structure relationship "/shared/sports-entertain.shtml → /shared/aboutua.shtml" emerged because of the website redesign, group II relationship between the two hyperlinks disappeared. Hence, LinkSelector did not pick /shared/aboutua.shtml using the July 2002 data.

Table 3.4: Hyperlinks Selected by LinkSelector (N=6)

No.	Sep. 2001	Feb. 2002	July 2002
1	/index/alldepts-index.shtml	/index/alldepts-index.shtml	/index/alldepts-index.shtml
2	/shared/sports-entertain.shtml	/shared/sports-entertain.shtml	/shared/sports-entertain.shtml
3	/working/teaching.shtml	/working/teaching.shtml	/working/teaching.shtml
4	/shared/aboutua.shtml	/shared/aboutua.shtml	/student_link
5	/shared/getting-around.shtml	/spotlight/index.shtml	/spotlight/index.shtml
6	/spotlight/index.shtml	/visiting/alumni.shtml	/newschedule/parse-schedule-new.cgi

Figure 3.9 and 3.10 depict the performance comparison among the three approaches using the Feb. 2002 testing data and the July 2002 testing data respectively.

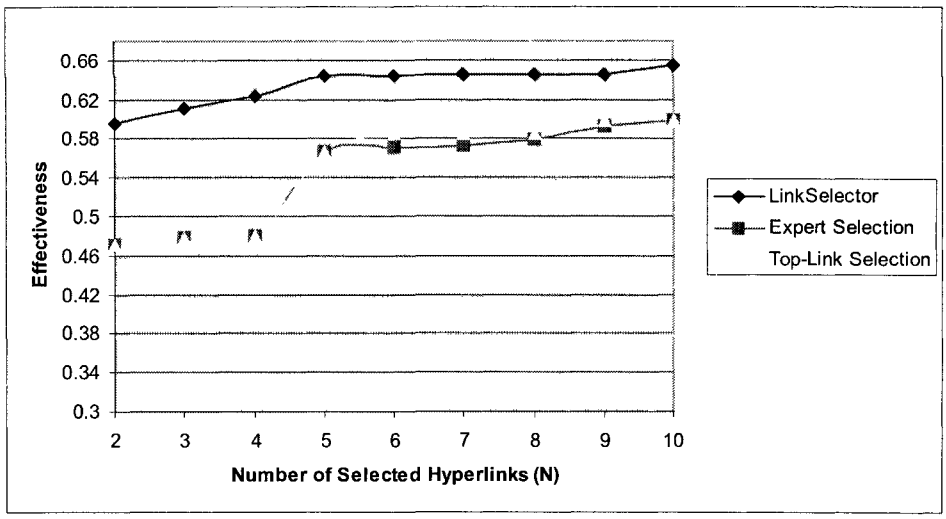


Figure 3.9: (a) Effectiveness Comparison Among LinkSelector, Expert Selection and Top-link Selection (Feb. 2002)

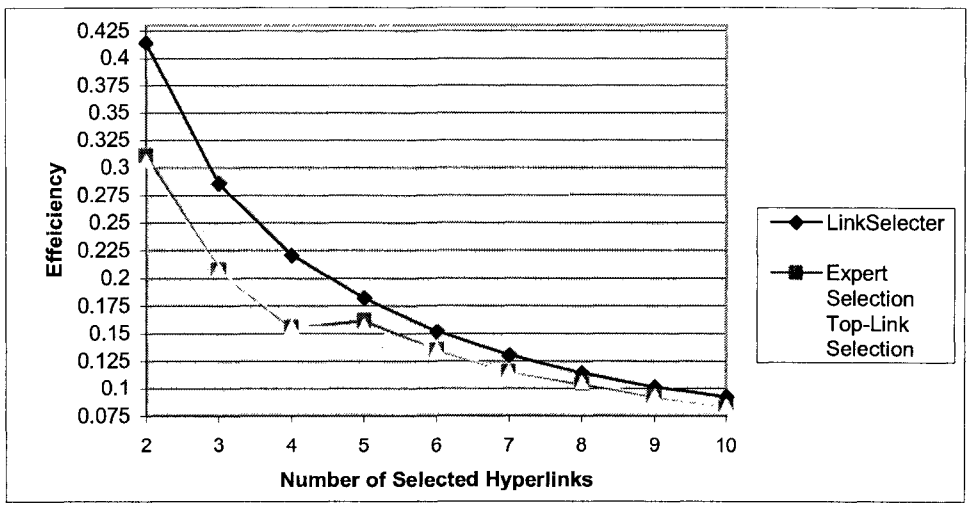


Figure 3.9: (b) Efficiency Comparison Among LinkSelector, Expert Selection and Top-link Selection (Feb. 2002)

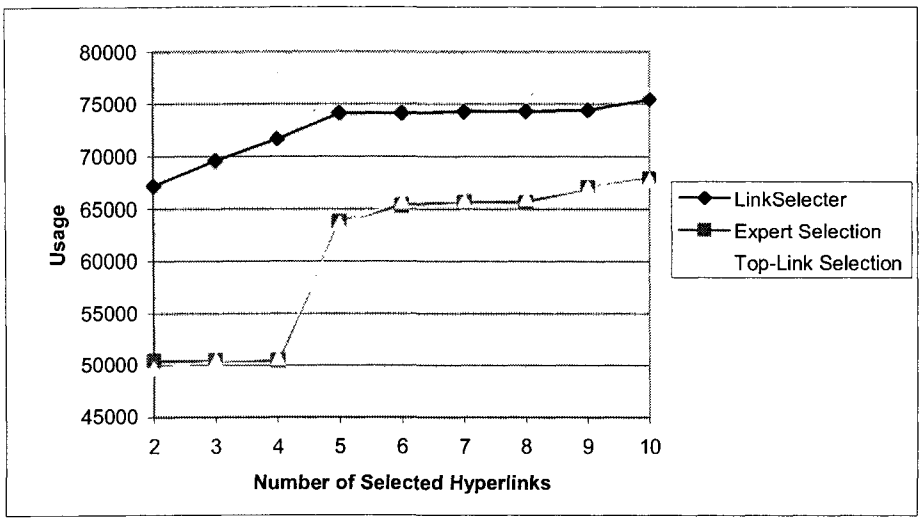


Figure 3.9: (c) Usage Comparison Among LinkSelector, Expert Selection and Top-link Selection (Feb. 2002)

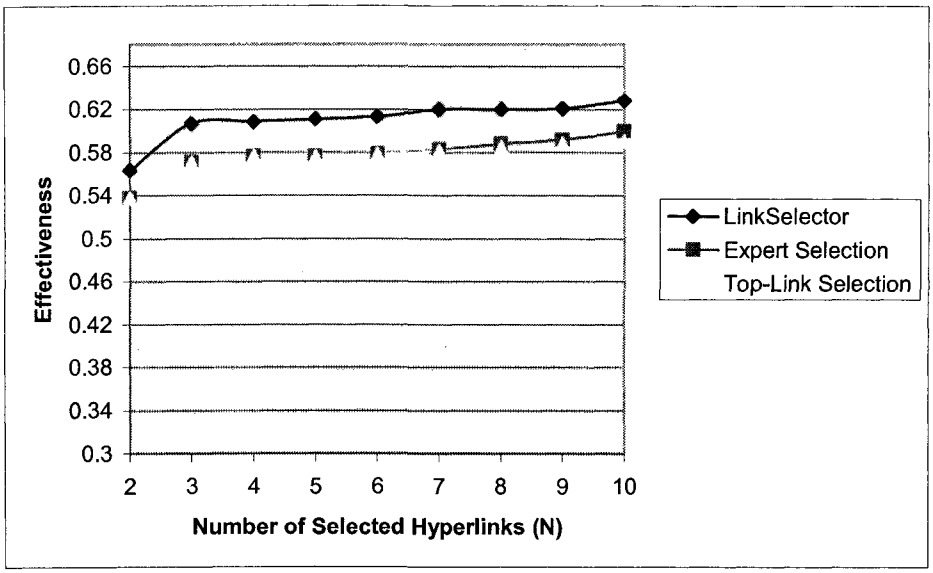


Figure 3.10: (a) Effectiveness Comparison Among LinkSelector, Expert Selection and Top-link Selection (July 2002)

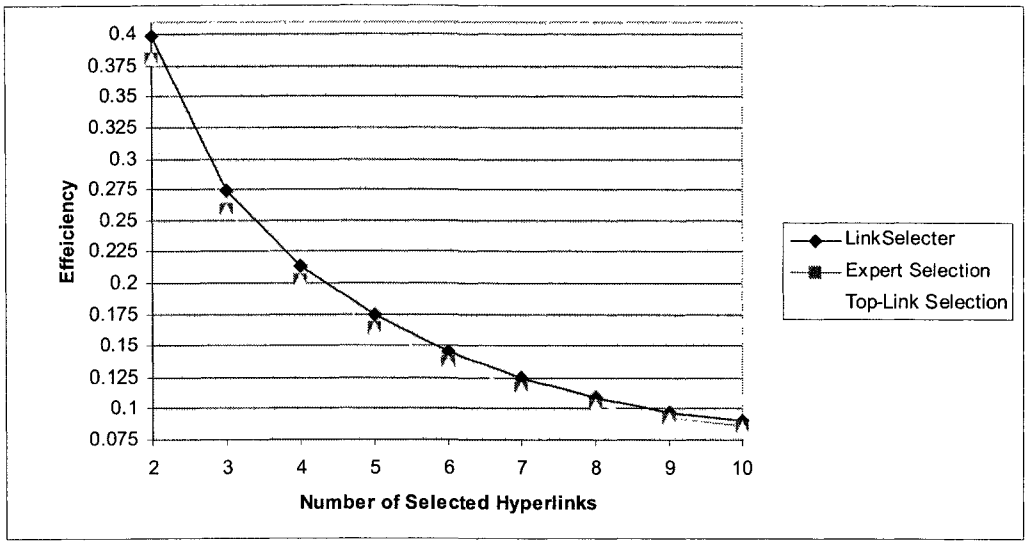


Figure 3.10: (b) Efficiency Comparison Among LinkSelector, Expert Selection and Top-link Selection (July 2002)

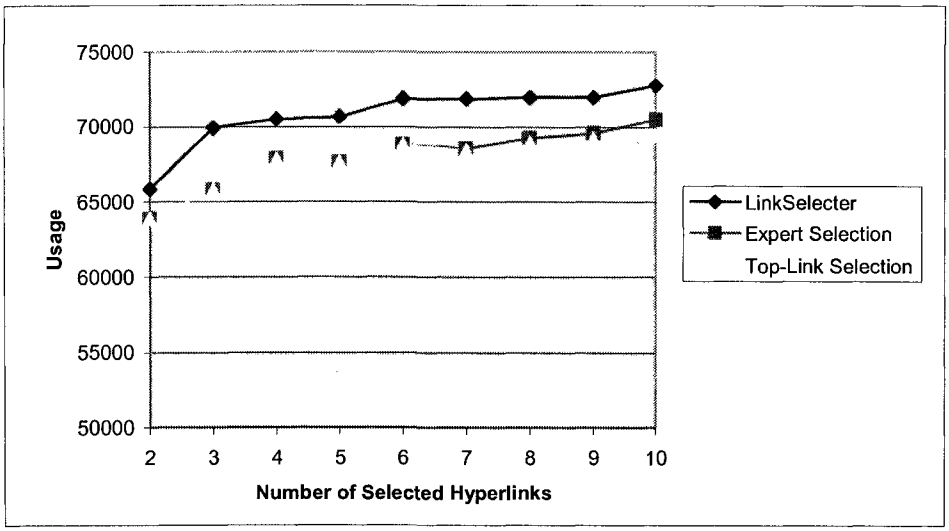


Figure 3.10: (c). Usage Comparison Among LinkSelector, Expert Selection and Top-link Selection (July 2002)

Using the Feb. 2002 data, compared with the other two approaches, LinkSelector improved effectiveness by an average of 16.2%, efficiency by an average of 20.1% and usage by an average of 21.3%. Using the July 2002 data, LinkSelector improved effectiveness by an average of 5.9%, efficiency by an average of 7.2% and usage by an average of 5.1%. In all the time periods, LinkSelector outperformed the other two approaches. The performance of expert selection and top-link selection was improved after the website redesign. The performance improvement was because some previously mentioned problems, such as the failure to consider group II relationship between /shared/sports-entertain.shtml and /shared/aboutua.shtml, were fixed after the website redesign (e.g., group II relationship between the above mentioned hyperlinks disappeared as the structure relationship between the hyperlinks emerged after the redesign). On the other hand, if LinkSelector had been adopted, these problems could have been fixed before the website redesign. Hence, the performance improvement also suggests that LinkSelector could be a good supporting tool in designing better portal pages. Even after the website redesign, the performance improvement of LinkSelector was still notable. On average 3068 more user-sought top level web pages could be easily accessed from the portal page constructed using LinkSelector than from those constructed utilizing the other two approaches. As the testing data covers a 7-day time period, average saving per day was 438. From the three experiments, we conclude that: (1) LinkSelector outperforms expert selection and top-link selection because of Group I and II relationships considered in LinkSelector but missed in the other two approaches; (2) the more Group I and II

relationships considered in LinkSelector but missed in the other two approaches the higher the performance improvement of LinkSelector.

3.6 Conclusion

In this chapter, we have formally defined the hyperlink selection problem and proposed a heuristic solution method named LinkSelector. The proposed method is based on relationships among hyperlinks – structure relationships extracted from an existing website and access relationships discovered from a web log. Preferences of hyperlinks and hyperlink sets are calculated from these relationships and a greedy algorithm is developed to extract hyperlinks from a given hyperlink pool using the preferences calculated. We also compare LinkSelector with the current practice of hyperlink selection and top-link selection, using data obtained from the University of Arizona website. Results show that LinkSelector, which integrates structure and access information of a website, outperforms the other hyperlink selection approaches.

A critical problem associated with LinkSelector is to make it adaptive to changes both in the structure of a website and in users' web visiting patterns. The former leads to changes in structure relationships and the latter causes changes in access relationships. As a result, hyperlinks selected by LinkSelector based on old structure relationships and access relationships could be out-of-date. To keep the selected hyperlinks up-to-date, an obvious solution is to re-run LinkSelector every time a change occurs. Apparently, for websites with frequent updates, the cost of frequent re-run is unbearable. The problem can be

categorized as a knowledge refreshing problem. We will address this problem in Chapter 5 based on the analytical model and its optimal solutions developed in Chapter 2.

CHAPTER 4: A DATA MINING BASED PREFETCHING APPROACH TO CACHING FOR NETWORK STORAGE SYSTEMS

In this chapter, we introduce another real world KDD application – a data mining based storage caching approach. Besides contributions to the area of intelligent caching, the application described in this chapter also provides a real world environment to study the knowledge refreshing problem. The majority of the chapter describes the data mining based caching approach. We briefly summarize knowledge refreshing requirements for the caching approach at the end of the chapter. Detailed knowledge refreshing requirements for the caching approach are illustrated and addressed in Chapter 5, based on the analytical model and its optimal solutions developed in Chapter 2.

4.1 Introduction

The need for network storage has been increasing at an exponential rate owing to the widespread use of the Internet in organizations and the shortage of local storage space due to the increasing size of applications and databases (Gibson and Meter 2000). Proliferation of network storage systems entails a significant increase in the amount of storage objects (e.g., files) stored, the number of concurrent clients, and the size and number of storage objects transferred between the systems and their clients. Performance (e.g., client perceived latency) of these systems becomes a major concern.

Previous research (Lee and Thekkath 1996, Thekkath et al. 1997) has explored techniques for scaling-up of the number of storage servers involved to enhance the performance of network storage systems. These techniques worked well for LAN-based network storage systems as increasing storage servers decreased the load for each server. However, adding servers to improve system performance is an expensive solution. Moreover, for a WAN-based network storage system, the bottleneck for its performance improvement typically is not caused by the load of storage servers but by the network traffic between clients and storage servers. A cost-effective way to improve its performance is to distribute storage servers and cache copies of storage objects at storage servers near the clients who request them (i.e., migrate copies of storage objects from storage servers that store them to storage servers near the clients who request them) (Gwertzman and Seltzer 1995, Barish and Obraczka 2000). This chapter introduces an Internet-based network storage system named NetShark and proposes a caching-based performance enhancement solution for such a system. The rest of the chapter is organized as follows. Features, implementation, architecture and performance measurements of NetShark are briefly described in Section 4.2. We review related research in Section 4.3 and propose a data-mining-based prefetching approach in Section 4.4. Section 4.5 presents the simulator used to evaluate the performance of the proposed caching approach. We summarize and discuss the simulation results in Section 4.6 and conclude the chapter in Section 4.7.

4.2 Overview of NetShark

In this section, we briefly introduce features, implementation, architecture and performance measurements of NetShark. Shark is a second generation IBM Enterprise Storage Server[13] with a maximum storage capacity of 11TB. NetShark is an Internet-based network storage system built on Sharks and provides storage services to users at both IBM and the University of Arizona. By providing university users with extra storage space through Internet, NetShark strengthens the shared IBM and university goals of providing better supports for online learning communities. Besides routine functionalities, such as file transfer, file management and user management, NetShark also provides file compression functionality, which allows users to compress files before transferring them, and file encryption functionality, which enables users to encrypt critical files. Figure 4.1 shows a screen shot of NetShark installed at the IBM Almaden Research Center.

[13] IBM Enterprise Storage Server is a copyrighted IBM product.

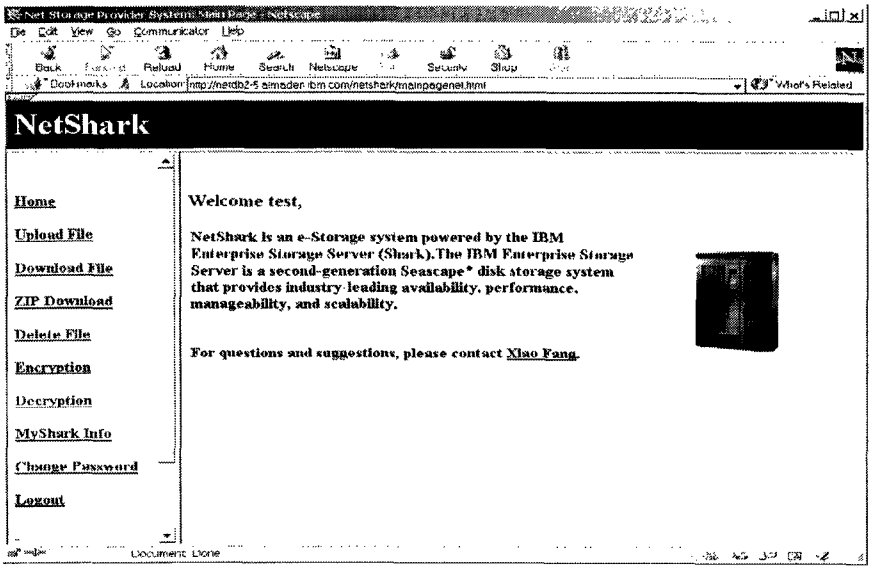


Figure 4.1: A Screen Shot of NetShark

As shown in Figure 4.2, the implementation of NetShark consists of three layers: presentation layer, application layer, and storage layer. NetShark was designed to be accessed easily anywhere in the world. Therefore, we chose web browsers (e.g., Netscape) as its presentation tool. Apache web server was selected as the web server of NetShark for handling HTTP requests from clients and delivering HTTP responses to clients. The application layer consists of Java servlets managed by WebSphere[14] and a database maintained by DB2[15], namely Log-DB. The application layer is responsible for user authentication, transaction processing and session management. In addition to web logs collected by the Apache web server, all transactions between clients and NetShark are recorded in Log-DB. Some critical information missing in web logs (Colley et al. 1999), such as session identification, are stored in Log-DB. Log-DB provides an ideal

[2,3] WebSphere and DB2 are copyrighted IBM products.

source for the data-mining-based prefetching approach to be described in Section 4.4. The storage layer is located in Sharks and communicates with the application layer through SCSI protocol.

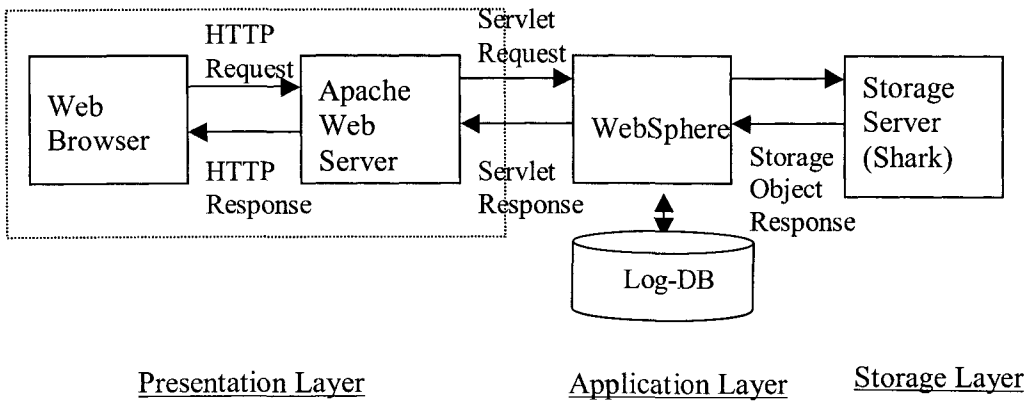


Figure 4.2: The Three-Layer Implementation Structure of NetShark

NetShark employs a geographically distributed network storage architecture in which Sharks are distributed in several geographically separated regions where their clients reside. We name a Shark placed in a client's region the Home Shark of the client. To reduce client perceived latency and to balance workloads among Sharks, clients' requests are always routed to and responded from their Home Sharks. If a storage object requested is not stored in a client's Home Shark, a copy of the object will be cached to the Home Shark from the Shark that contains the object (hereafter called the Storage Shark).

Figure 4.3 offers an example of the NetShark architecture. In this example, NetShark hosts a Knowledge Management System (KMS) which consists of modular knowledge objects in various file and storage formats. Clients of the KMS are distributed in three geographically separated regions, New York, San Jose and Tucson. Sharks distributed in New York, San Jose and Tucson are denoted as Shark NYC, Shark SJC and Shark TUS respectively. A logical scenario of storage distribution could allocate the operating system knowledge objects to Shark NYC, the database knowledge objects to Shark SJC and the storage server knowledge objects to Shark TUS. In addition to serving as the Storage Sharks of the KMS, these three Sharks are also the designated Home Sharks for the clients in their located regions. When clients in Tucson request remote objects stored at Shark SJC, the requested objects will be cached from Shark SJC to the clients' Home Shark, Shark TUS, and remain in Shark TUS until other cached objects replace them. In this example, clients connect to their Home Sharks through Intranet/Internet and Sharks are connected using Public Switched Data Network (PSDN), a popular and high-performance way to connect servers in different regions.

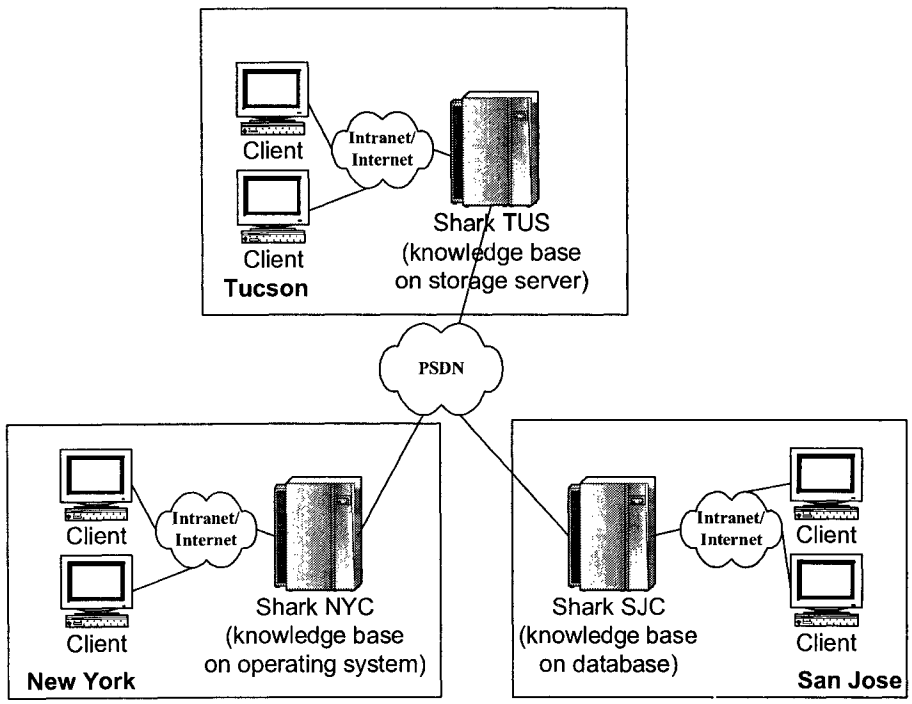


Figure 4.3: The Geographically Distributed Network Storage Architecture of NetShark

To compare different caching approaches for NetShark, we introduced three performance measurements for NetShark: client perceived latency, hit rate and Shark-Shark response time. Client perceived latency and hit rate are popular performance measurements for web caching.

Measurement 1: Client perceived latency[16] refers to the delay between the time when a client submits a storage object request and the time when the storage object is received by the client.

Retrieving storage objects directly from clients' Home Sharks saves the time of transferring these storage objects from their Storage Sharks to clients' Home Sharks. Therefore, increasing the number of storage object requests that can be directly fulfilled from Home Sharks reduces client perceived latency.

Measurement 2: Hit rate, $hr = \frac{n_{home}}{n_{req}}$, is used to measure the capability that Home Sharks can directly respond to storage object requests. Here n_{home} is the number of storage object requests that can be directly fulfilled from Home Sharks and n_{req} is the total number of storage object requests.

Measurement 3: To measure the impact of network traffic between Sharks, Shark-Shark response time is the length of the interval between the time when a Shark submits a storage object request to a remote Shark and the time when the storage object is received by the requesting Shark.

[16] Client perceived latency is equivalent to response time in the distributed computing literatures.

4.3 Related Work

In this section, we review research on web caching[17] on which our caching solution is based. Web caching is the temporary storage of web objects for later retrieval (Barish and Obraczka 2000). Major issues in web caching include what web objects need to be cached, how to replace old web objects when there is not enough space for newly cached web objects and how to keep consistency among copies of web objects. According to the aforementioned issues, web caching research can be divided into three parts: caching, replacement and consistency.

4.3.1 Caching

A simple caching approach, namely caching on demand, only caches currently requested web objects. More efficient caching approaches, prefetching approaches, cache both currently requested web objects and web objects predicted to be requested in the near future (Kroeger et al. 1997). Popularity-based prefetching approaches (Dias et al. 1996, Markatos and Chronaki 1998, Kim et al. 2000) predicted and prefetched future requested web objects based on their past request frequencies. Another type of prefetching approaches discovered and utilized access relationships between web objects in making prefetching decisions (Padmanabhan and Mogul 1996, Horng et al. 1998, Fan et al. 1999, Lou and Lu 2002). For example, Padmanabhan and Mogul (1996) modeled access

[17] There have been research on file migration for hierarchical storage management and distributed systems since 1970s (e.g., Liu Sheng 1992). Lots of past research results have been incorporated into web caching. Here, we present a review on web caching research related to this chapter.

relationships between web objects using a dependency graph. In the dependency graph, nodes represented web objects and arcs between nodes represented access relationships between web objects (i.e., how likely one web object will be requested after another web object). Lou and Lu (2002) extended the work in (Padmanabhan and Mogul 1996) and considered not only access relationships between web objects but also access relationships between web sites when making prefetching decisions. Chinen and Yamaguchi (1997) proposed a different approach to predicting and prefeching future requested web objects based on structural relationships between web objects. The proposed approach parsed HTML files and prefetched embedded images and web objects pointed to by embedded links. However, the approach greatly increased network traffic between servers and proxies and it is not practical to prefetch web objects solely using the approach. The prefetching approach proposed in (Davison 2002) predicted next requested web page by analyzing the content of the web pages requested recently. It had been shown in (Padmanabhan and Mogul 1996, Crovella and Barford 1998) that prefetching approaches increased network traffic between servers and proxies, which could impact client perceived latency negatively. However, none of the past prefetching approaches has addressed the network traffic increase problem. In (Padmanabhan and Mogul 1996, Horng et al. 1998, Fan et al. 1999), access relationships between web objects were extracted based on an arbitrarily chosen look-ahead window and these approaches neglected session – a natural unit to extract access relationships between web objects. Aware of the limitations of the past prefetching approaches, the proposed prefetching approach will:

- predict and prefetch future requested objects based on sequential object request patterns within sessions and between sessions;

- divide future requested objects into two categories, urgent and wait, and cache objects in the urgent category immediately while cache objects in the wait category when the network traffic is below a user-defined threshold.

4.3.2 Replacement

Least-Recently-Used (LRU) and Least-Frequently-Used (LFU) are two widely used cache replacement approaches. LRU is concerned with recency of object requests, while LFU is concerned with frequency of object requests. LRU, which was previously used for page replacement in memory management (Tanenbaum and Woodhull 1997), is based on the observation that web objects that have been heavily requested recently will probably be heavily requested in the near future. Conversely, web objects that have not been used for ages will probably remain unused for a long time. Hence, the web object that have not been used for the longest time is replaced by LRU. LFU replaces the web object with the least request frequency. Cache replacement approaches proposed in (Cao and Irani 1997, Jin and Bestavros 2000) considered the cost of transferring a copy of web object from a server to a proxy, namely cache cost. Obviously, it is desirable to replace the web object with the lowest cache cost, if everything else is equal. In (Chang and Chen 2002), the replacement decision is based on a profit function of objects and the profit function is defined based on expected request frequencies of objects and sizes of objects etc. We

adopt LRU, a widely used replacement approach, in NetShark. Description of LRU can be found in lots of literatures, such as (Tanenbaum and Woodhull 1997), and is not included.

4.3.3 Consistency

There are two types of cache consistency approaches: weak cache consistency and strong cache consistency (Cao and Liu 1998). Weak cache consistency approaches might return stale web objects to users while strong cache consistency approaches guarantee to return up-to-date web objects to users. Client polling and TTL (i.e., time to live) are weak cache consistency approaches (Barish and Obraczka 2000). With client polling, cached objects are periodically compared with their original copies. Out-of-date cached objects are dumped and their newest versions are fetched. In TTL, each cached object has a time to live (TTL). When expired, these objects are discarded and their newest versions are fetched. Invalidation callback is a strong cache consistency approach (Barish and Obraczka 2000). It requires a server to keep track of all the cached objects. The server will notify all the proxies to invalidate their copies if the original object has been modified in the server. It has been shown in (Cao and Liu 1998) that strong cache consistency approaches can be realized with no or little extra cost than weak cache consistency approaches. In addition, in a recent study by Yin et al. (2002), strong cache consistency approaches increase hit rate by a factor of 1.5-3 for large-scale dynamic web sites, compared with weak cache consistency approaches. We adopt a strong cache consistency approach, the invalidation callback approach, to maintain consistency among

storage objects in NetShark. Description of the approach can be found in (Barish and Obraczka 2000) and is not included.

4.4 A Data-Mining-Based Prefetching Approach

Similar to web caching, major issues in network storage caching include what storage objects need to be cached (i.e., caching), how to replace old storage objects when there is not enough space for newly cached storage objects (i.e., replacement) and how to keep consistency among copies of storage objects (i.e., consistency). For NetShark, we propose a data-mining-based prefetching approach as the caching approach and we adopt LRU and invalidation callback as replacement and consistency approaches respectively.

The proposed prefetching approach consists of two algorithms: offline learning and online caching. Client request patterns are extracted from Log-DB periodically, using the offline learning algorithm. Based on the patterns extracted, the online caching algorithm caches storage objects. Table 4.1 summarizes the important notations to be referenced in Sections 4.4 and 4.5.

Table 4.1: Notation Summary

Notation	Description
p_{intra}, p^{i}_{intra}	an intra-session pattern
P_{intra}	a set of intra-session patterns, where $p_{intra} \in P_{intra}$
$S(p_{intra})$	the set of all storage objects in p_{intra}
p_{inter}, p^{i}_{inter}	an inter-session pattern

P_{inter}	a set of inter-session patterns, where $p_{inter} \in P_{inter}$
$S(p_{inter})$	the set of all storage object sets in p_{inter}
SO, SO_i	a set of storage objects
so_d	the storage object on demand
so_r	a related storage object of so_d, predicted from intra/inter-session patterns
SO_S	the set of storage objects requested so far in a session , where $so_d \in SO_S$
$pattern(SO_S)$	an intra/inter-session pattern that contains SO_S
$Pattern(SO_S)$	the set of all intra/inter-session patterns that contain SO_S
$ti(so_r)$	the predicted time interval between so_d and so_r
$\sup(so_r)$	the support of so_r, which measures how likely so_r will be requested after so_d
$nw(so)$	the network over which a storage object, so, is transferred
$transfer(so, nw(so))$	the time to transfer a storage object, so, over network $nw(so)$, which can be calculated using (1)
$HD(nw(so))$	the average hop delay of network $nw(so)$
$HN(nw(so))$	the number of hops on network $nw(so)$
$D(nw(so))$	the distance of network $nw(so)$
$v(nw(so))$	the signal velocity of network $nw(so)$
$B(nw(so))$	the bandwidth of network $nw(so)$
$size(so)$	the size of a storage object , so
$slack(so_r)$	the slack value of so_r, which can be calculated using (2)
n_{NS}	the total number of storage objects in NetShark
SO_{NS}	the set of all storage objects in NetShark
$fre(so_i)$	the request frequency rate of so_i, where $so_i \in SO_{NS}$ for $i = 1,2,\cdots n_{NS}$
$fanout(so_i)$	the fan-out number of so_i, where $so_i \in SO_{NS}$ for $i = 1,2,\cdots n_{NS}$
$random(so_i)$	the random factor of so_i, where $so_i \in SO_{NS}$ for $i = 1,2,\cdots, n_{NS}$

4.4.1 The Offline Learning Algorithm

We briefly illustrate the structure of Log-DB before introducing the offline learning algorithm. A record in Log-DB has the following fields: ClientID, ClientIP, SessionID, FileName, FileType, FileSize and RequestTime. As shown in Table 4.2, critical fields for offline learning include ClientID, SessionID, FileName, and RequestTime, which respectively specify who requested which storage object (e.g., file) at what time and in which session. Here, a session is a sequence of storage object requests between a client's log-in and log-out of NetShark. It can be easily seen from Table 4.2 that a client (e.g., *CT1*) may have several sessions (e.g., *SN1* and *SN2*) and a session (e.g., *SN1*) may include several storage object requests (e.g., *A*, *B* and *C*).

Table 4.2: Critical Fields in Log-DB

ClientID	SessionID	FileName	RequestTime
CT1	*SN1*	*A*	*09:19:34 07/25/2001*
CT1	*SN1*	*B*	*09:20:34 07/25/2001*
CT1	*SN1*	*C*	*09:24:34 07/25/2001*
CT1	*SN2*	*F*	*10:57:34 07/25/2001*
CT1	*SN2*	*E*	*11:01:34 07/25/2001*
CT2	*SN3*	*F*	*09:57:34 07/25/2001*
CT2	*SN3*	*D*	*10:00:34 07/25/2001*
CT3	*SN4*	*A*	*09:29:34 07/25/2001*
CT3	*SN4*	*B*	*09:30:34 07/25/2001*
CT3	*SN4*	*C*	*09:32:34 07/25/2001*
CT3	*SN5*	*D*	*11:07:34 07/25/2001*
CT3	*SN5*	*E*	*11:09:34 07/25/2001*

Instead of an arbitrarily chosen lookahead window used in (Padmanabhan and Mogul 1996), the offline learning algorithm uses session as the basic processing unit to discover client request patterns. Two types of the patterns can be extracted from Log-DB: intra-session patterns P_{intra} and inter-session patterns P_{inter}.

Definition 4.1: An intra-session pattern, p_{intra}, where $p_{intra} \in P_{intra}$, reflects a request behavior within a session. It has the format: $\langle so_1 \xrightarrow[\text{time interval}]{} so_2 \cdots\cdots so_{n-1} \xrightarrow[\text{time interval}]{} so_n \text{ support} \rangle$. Here $so_1, so_2, \cdots, so_n$ are storage objects requested within the same session. The time interval between two storage objects is the average time interval between the requests of the two storage objects in sessions where the pattern can be found. The support of an intra-session pattern is the fraction of sessions in which the pattern can be found.

For an intra-session pattern, p_{intra}, with the format $\langle so_1 \xrightarrow[\text{time interval}]{} so_2 \cdots\cdots so_{n-1} \xrightarrow[\text{time interval}]{} so_n \text{ support} \rangle$, we denote $S(p_{intra})$ as the set of all storage objects in it, i.e., $S(p_{intra}) = \{so_i\}$ for $i = 1,2,\cdots,n$.

Example 4.1: An intra-session pattern, $p_{intra} = \langle A \xrightarrow[\text{1 minute}]{} B \xrightarrow[\text{3 minutes}]{} C \ 0.4 \rangle$, can be discovered from the Log-DB example in Table 4.2. According to this pattern, storage objects A, B and C are sequentially requested within a session; such a pattern can be found in 40% (i.e., *SN1* and *SN4*) of the total sessions; and the average time intervals

between the requests of A and B and between the requests of B and C are 1 minute and 3 minutes respectively. In this example, $S(p_{intra}) = \{A, B, C\}$.

Definition 4.2: An inter-session pattern, p_{inter}, where $p_{inter} \in P_{inter}$, reveals a request behavior between sessions. It has the format: $\langle SO_1 \xrightarrow[\text{time interval}]{} SO_2 \cdots\cdots SO_{n-1} \xrightarrow[\text{time interval}]{} SO_n \text{ support} \rangle$. Here $SO_1, SO_2 \cdots, SO_n$ are storage object sets requested in different sessions by the same client. The time interval between two storage object sets is the average time interval between the requests of the two storage object sets among clients demonstrating such a pattern. The support of an inter-session pattern is the fraction of clients demonstrating such a pattern.

For an inter-session pattern, p_{inter}, with the format $\langle SO_1 \xrightarrow[\text{time interval}]{} SO_2 \cdots\cdots SO_{n-1} \xrightarrow[\text{time interval}]{} SO_n \text{ support} \rangle$, we denote $S(p_{inter})$ as the set of all storage object sets in it, i.e., $S(p_{inter}) = \{SO_i\}$ for $i = 1,2,\cdots,n$.

Example 4.2: An inter-session pattern, $p_{inter} = \langle \{A, B\} \xrightarrow[100 \text{ minutes}]{} \{E\}\ 0.66 \rangle$, can be learned from the Log-DB example in Table 4.2. According to this pattern, storage object sets $\{A,B\}$ and $\{E\}$ are sequentially requested in different sessions by the same client; 66% (i.e., *CT1* and *CT3*) of total clients demonstrate such a pattern; and the average time interval between the request of $\{A,B\}$ and the request of $\{E\}$ is 100 minutes (i.e., the average time interval between the request of B, the last requested storage object in $\{A,B\}$,

and the request of E, the first requested storage object in $\{E\}$). In this example, $S(p_{inter}) = \{\{A,B\},\{E\}\}$.

Both intra-session patterns and inter-session patterns can be extracted from Log-DB using sequential pattern mining. Agrawal and Srikant (1995) used a database of customer transactions to define sequential pattern mining. In this database, a customer had k transactions, where $k \geq 1$, and a transaction included the purchases of m items, where $m \geq 1$. As shown in the example in Table 4.3, each record in the database consists of customer-id, transaction-id, item-id and transaction-time. The database is sorted by increasing customer-id and then by increasing transaction-time.

Table 4.3: Customer Transactions For Sequential Pattern Mining

customer-id	transaction-id	item-id	transaction-time
CUST1	*T1*	*1*	*09:19:34 07/25/2001*
CUST1	*T1*	*2*	*09:19:34 07/25/2001*
CUST1	*T1*	*4*	*09:19:34 07/25/2001*
CUST1	*T2*	*5*	*11:21:34 07/26/2001*
CUST1	*T2*	*3*	*11:21:34 07/26/2001*
CUST2	*T3*	*1*	*09:29:34 07/25/2001*
CUST2	*T3*	*2*	*09:29:34 07/25/2001*
CUST2	*T4*	*6*	*10:21:34 07/28/2001*
CUST2	*T4*	*3*	*10:21:34 07/28/2001*
CUST3	*T5*	*8*	*13:21:34 07/30/2001*
CUST3	*T5*	*3*	*13:21:34 07/30/2001*

In sequential pattern mining, an itemset is a non-empty set of items. A sequence is an ordered list of itemsets and it reveals an inter-transaction purchasing behavior among

customers. For example, {1,2} is an itemset and ⟨{1,2}{3}⟩ is a sequence with the meaning that item 3 was purchased in a transaction after a transaction that had purchased items 1 and 2. A customer supports a sequence if and only if the sequence can be found in the transactions of the customer. The support of a sequence is defined as the fraction of total customers who support the sequence. For example, sequence ⟨{1,2}{3}⟩ is supported by customers *CUST1* and *CUST2* in the example given in Table 4.3 and its support is 66%. Sequential pattern mining is conducted to discover all large sequences, which are sequences with support larger than a user-defined threshold.

Before extracting intra/inter-session patterns from Log-DB, the database is sorted first by increasing ClientID and then by increasing RequestTime. Table 4.2 gives an example of a sorted Log-DB. To extract inter-session patterns from Log-DB, large sequences are discovered using sequential pattern mining. In this case, ClientID, SessionID and FileName in Log-DB (see Table 4.2) correspond to customer-id, transaction-id and item-id (see Table 4.3) respectively. Every large sequence discovered from Log-DB is an ordered list of storage object sets and it reveals an inter-session request behavior among clients. During the process of discovering large sequences, time intervals between storage object sets in sequences are calculated and incorporated into these sequences. The discovered large sequences with calculated time intervals are inter-session patterns.

To discover intra-session patterns, we create pseudo-transactions by assigning a Pseudo-TransactionID for each storage object request (i.e., each record) in Log-DB, as shown in

Table 4.4. Similarly, large sequences are discovered using sequential pattern mining. In this case, SessionID, Pseudo-TransactionID and FileName in Log-DB (see Table 4.4) correspond to customer-id, transaction-id and item (see Table 4.3) respectively. Every large sequence discovered is an ordered list of storage object sets with only one element because each pseudo-transaction has only one storage object request. Large sequences discovered reveal inter-pseudo-transaction request behaviors among sessions (i.e., intra-session request behaviors). Similarly, time intervals between storage objects in sequences are calculated and incorporated into these sequences. The discovered sequences with calculated time intervals are intra-session patterns.

Table 4.4: Add the Pseudo-TransactionID Field Into Log-DB

ClientID	SessionID	Pseudo-TransactionID	FileName	RequestTime
CT1	*SN1*	*1*	*A*	*09:19:34 07/25/2001*
CT1	*SN1*	*2*	*B*	*09:20:34 07/25/2001*
CT1	*SN1*	*3*	*C*	*09:24:34 07/25/2001*
CT1	*SN2*	*4*	*F*	*10:57:34 07/25/2001*
CT1	*SN2*	*5*	*E*	*11:01:34 07/25/2001*
CT2	*SN3*	*6*	*F*	*09:57:34 07/25/2001*
CT2	*SN3*	*7*	*D*	*10:00:34 07/25/2001*
CT3	*SN4*	*8*	*A*	*09:29:34 07/25/2001*
CT3	*SN4*	*9*	*B*	*09:30:34 07/25/2001*
CT3	*SN4*	*10*	*C*	*09:32:34 07/25/2001*
CT3	*SN5*	*11*	*D*	*11:07:34 07/25/2001*
CT3	*SN5*	*12*	*E*	*11:09:34 07/25/2001*

We summarize the offline learning algorithm in Figure 4. We denote the number of records in Log-DB as n_{rec}. The offline learning algorithm includes four parts: sorting Log-DB, assigning Pseudo-TransactionID, discovering P_{inter} and discovering P_{intra}.

Sorting Log-DB needs $O(n_{rec} \log n_{rec})$ time. Assigning Pseudo-TransactionID requires $O(n_{rec})$. During the discovery of P_{inter}, sequential pattern mining goes through Log-DB α rounds, where α is determined by the maximum size of storage object sets and the maximum size of large sequences (Agrawal and Srikant 1995) [18]. Therefore, the time complexity of discovering P_{inter} is $O(n_{rec} \times \alpha)$. Usually, α is much less than n_{rec}. Hence, the time complexity of discovering P_{inter} is $O(n_{rec})$. Similarly, the time complexity of discovering P_{intra} is $O(n_{rec})$. Combining the time complexities of all these parts, the time complexity of the offline learning algorithm is $O(n_{rec} \log n_{rec})$. The main I/O cost of the algorithm is the I/O of Log-DB. Hence, the I/O cost of the algorithm is determined by such factors as the maximum size of storage object sets and the maximum size of large sequences.

Input: Log-DB
Output: P_{intra} : intra-session patterns,
P_{inter} : inter-session patterns.

sort Log-DB by increasing ClientID and then by increasing RequestTime;
assign a Pseudo-TransactionID for each storage object request (i.e., record) in Log-DB;
apply sequential pattern mining to Log-DB to discover P_{inter};
apply sequential pattern mining to Log-DB to discover P_{intra};

Figure 4.4 : The Offline Learning Algorithm

[18] The maximum size of storage object sets and the maximum size of large sequences are determined by the minimum support threshold picked for sequential pattern mining.

4.4.2 The Online Caching Algorithm

A storage object on demand, so_d, is one that currently has been requested by a client. The online caching algorithm caches so_d as well as its related storage objects, $\{so_r\}$ predicted from intra-session patterns (e.g., storage objects predicted to be requested right after so_d within a session) and inter-session patterns (e.g., storage objects predicted to be requested in a session after the session requesting so_d). Although prefetching algorithms are more efficient than caching on demand (Kroeger et al. 1997), these algorithms have a well-known shortcoming of increasing network traffic between servers and proxies (e.g., between Sharks in NetShark) compared with caching on demand (Padmanabhan and Mogul 1996). The online caching algorithm addresses the network traffic increase problem by dividing related storage objects $\{so_r\}$ into two categories, urgent and wait, based on time intervals in intra/inter-session patterns. Related storage objects in the urgent category, denoted as $\{so_{r_u}\}$, where $\{so_{r_u}\} \subseteq \{so_r\}$, are cached immediately, while related storage objects in the wait category, denoted as $\{so_{r_w}\}$, where $\{so_{r_w}\} \subseteq \{so_r\}$, are cached when the network traffic between Home and Storage Sharks is below a user-defined threshold.

As shown in Figure 4.5, the online caching algorithm consists of five parallel procedures: Process_Requests, Predict_Related_Objects, Process_Urgent_Objects, Process_Wait_Objects and Transfer_Objects.

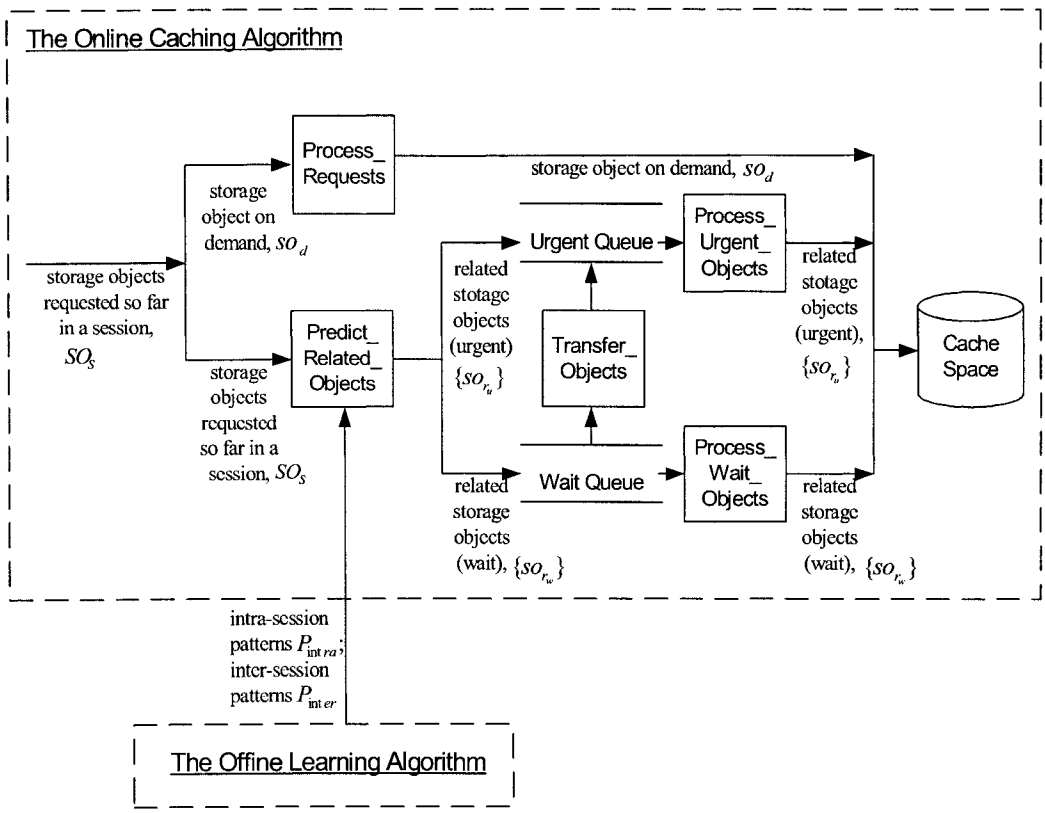


Figure 4.5: The Online Caching Algorithm

Procedure Process_Requests is triggered whenever there is a storage object request. It is responsible for returning the storage object on demand, so_d, to the requesting client. It also caches so_d if it is not in the Home Shark of the requesting client.

Procedure Predict_Related_Objects is also triggered by a storage object request. Input of the procedure includes the set of storage objects requested so far in a session, SO_S, which contains the storage object on demand, so_d, i.e. $so_d \in SO_S$, intra-session patterns P_{intra} and inter-session patterns P_{inter}. The procedure first predicts related storage objects

$\{so_r\}$ of so_d by searching for intra-session patterns P_{intra} and inter-session patterns P_{inter} that contain SO_S.

Definition 4.3: SO_S is *contained* in an intra-session pattern, p_{intra}, iff,

- $SO_S \subset S(p_{intra})$;
- and the request sequence of storage objects in SO_S is the same as their request sequence in p_{intra}.

Definition 4.4: SO_S is *contained* in an inter-session pattern, p_{inter}, iff $SO_S \in S(p_{inter})$.

We denote an intra/inter-session pattern that contains SO_S as $pattern(SO_S)$ and the set of all intra/inter-session patterns that contain SO_S as $Pattern(SO_S)$. The storage objects requested right after SO_S in $Pattern(SO_{cs})$ are related storage objects $\{so_r\}$ of so_d. Two attributes of a related storage object, so_r, $ti(so_r)$ and $\sup(so_r)$, can also be predicted from $Pattern(SO_S)$. $ti(so_r)$ is the predicted time interval between so_d and so_r, which is the time interval between so_d and so_r in the $pattern(SO_S)$ that includes so_r. $\sup(so_r)$ is the support of so_r, which measures how likely so_r will be requested after so_d. $\sup(so_r)$ is the support of the $pattern(SO_S)$ that includes so_r.

Example 4.3: Given the following intra/inter-session patterns:

$$p^0_{intra}: \langle B \xrightarrow[2\text{ minutes}]{} A \xrightarrow[5\text{ seconds}]{} I \ 0.08 \rangle,$$

$$p^1_{intra}: \langle A \xrightarrow[1\text{ minute}]{} B \xrightarrow[5\text{ seconds}]{} C \xrightarrow[1\text{ minute}]{} K \ 0.12 \rangle,$$

$$p^2_{intra}: \langle A \xrightarrow[1\text{ minute}]{} B \xrightarrow[4\text{ minutes}]{} F \ 0.1 \rangle,$$

$$p^0_{inter}: \langle \{A, B\} \xrightarrow[100\text{ minutes}]{} \{E, H\} \ 0.12 \rangle, \text{ and}$$

$$p^1_{inter}: \langle \{D, F\} \xrightarrow[10\text{ minutes}]{} \{G\} \ 0.1 \rangle,$$

if $SO_S = \{A, B\}$ $so_d = B$ and the request sequence of storage objects in SO_S is first A then B (i.e., $A \rightarrow B$), then SO_S is contained in p^1_{intra} by Definition 4.3, as,

- $SO_S \subset S(p^1_{intra})$, where $S(p^1_{intra}) = \{A, B, C, K\}$;
- the request sequence of storage objects in SO_S, $A \rightarrow B$, is the same as their request sequence in p^1_{intra}.

Similarly, SO_S is also contained in p^2_{intra} by Definition 3. SO_S is contained in p^0_{inter} by Definition 4.4, as $SO_S \in S(p^0_{inter})$, where $S(p^0_{inter}) = \{\{A, B\}, \{E, H\}\}$. Therefore, $Pattern(SO_S) = \{p^1_{intra}, p^2_{intra}, p^0_{inter}\}$. The storage objects requested right after SO_S in

p^1_{intra}, p^2_{intra} and p^0_{inter} are C, F, E and H. Hence, $\{so_r\} = \{C, F, E, H\}$. From $Pattern(SO_S)$, we also get,

so_r	$ti(so_r)$	$\sup(so_r)$
C	5 seconds	0.12
F	4 minutes	0.1
E	100 minutes	0.12
H	100 minutes	0.12

Related storage objects $\{so_r\}$ are distributed into the urgent queue or the wait queue based on their slack values $slack(so_r)$. We introduce the follows before defining $slack(so_r)$. We denote $nw(so)$ as the network over which a storage object, so, is transferred. The time to transfer so over network $nw(so)$, $transfer(so, nw(so))$, can be calculated using the following formula (Dalley 2001).

$$transfer(so, nw(so)) = HD(nw(so)) \times HN(nw(so)) + \frac{D(nw(so))}{v(nw(so))} + \frac{size(so) \times 8}{B(nw(so))} \quad (4.1)$$

Here $HD(nw(so))$ is the average hop delay of network $nw(so)$, $HN(nw(so))$ is the number of hops on network $nw(so)$, $D(nw(so))$ is the distance of network $nw(so)$, $v(nw(so))$ is the signal velocity of network $nw(so)$ (see Table 4.5 for some examples of $v(\cdot)$), $B(nw(so))$ is the bandwidth of network $nw(so)$ and $size(so)$ is the size of so.

Table 4.5: Signal Velocity of Different Media (Steinke 2001)

Medium	Signal velocity (km/second)
Phone line	160935
Copper (category 5)	231000
Optical fiber	205000
Air	299890

$slack(so_r)$ is defined using the following formula.

$$slack(so_r) = ti(so_r) - transfer(so_r, nw(so_r)) \quad (4.2)$$

$nw(so_r)$ is the network over which so_r is transferred. In (4.2), $nw(so_r)$ refers to the network between the Storage Shark of so_r and the Home Shark of the client who could request so_r. $slack(so_r)$ reflects the degree of urgency to cache a related storage object so_r. Related storage objects with slack values less than or equal to 0 are placed in the urgent queue and need to be cached immediately; related storage objects with slack values larger than 0 are placed in the wait queue and cached when the network traffic between Home and Storage Sharks is below a user-defined threshold. Related storage objects in both urgent queue and wait queue are sorted by their descending supports (i.e., $\sup(so_r)$) then by their ascending slack values (i.e., $slack(so_r)$).

Example 4.4: For related storage objects, C, E, F and H, predicted in Example 4.3, $size(C) = 500\text{k bytes}, size(E) = 50\text{k bytes}, size(F) = 100\text{k bytes}, \quad size(H) = 100\text{k bytes}$. Given the network over which these storage objects transferred, $nw(so_r)$, and its

characteristics: $HD(nw(so_r)) = 0.015$ second, $HN(nw(so_r)) = 18$, $D(nw(so_r)) = 1000$ km, $v(nw(so_r)) = 231000$ km/second, $B(nw(so_r)) = 500$k bps, slack values of these storage objects can be calculated as follows:

$$transfer(C, nw(so_r)) = 0.015 \times 18 + \frac{1000}{231000} + \frac{500\text{k} \times 8}{500\text{k}} = 8.2743 \text{ seconds}$$

$$slack(C) = ti(C) - transfer(C, nw(so_r)) = 5 - 8.2743 = -3.2743 \text{ seconds}$$

Similarly,

$$slack(E) = ti(E) - transfer(E, nw(so_r)) = 5998.926 \text{ seconds}$$

$$slack(F) = ti(F) - transfer(F, nw(so_r)) = 238.1257 \text{ seconds}$$

$$slack(H) = ti(H) - transfer(H, nw(so_r)) = 5998.126 \text{ seconds}$$

According to the slack values calculated, related storage object C is placed in the urgent queue while related storage objects E, F and H are put in the wait queue.

Procedure Predict_Related_Objects is summarized in Figure 4.6 below. The time complexity of the procedure is $O(n_{intra} + n_{inter})$, where n_{intra} is the number of intra-session patterns and n_{inter} is the number of inter-session patterns.

```
Event: a storage object request /* event that triggers the procedure*/
Input: P_intra : intra-session patterns,
       P_inter : inter-session patterns,
       SO_S : the set of storage objects requested so far in a session,
            where so_d ∈ SO_S .

For each intra-session pattern p_intra , where p_intra ∈ P_intra
    If ( SO_S is contained in p_intra )
        so_r = the storage object in p_intra requested right after SO_S ;
        If ( so_r not in clients' Home Shark)
            calculate slack(so_r) ;
            If ( slack(so_r) ≤ 0))
                urgent.add( so_r ); /* add so_r into the urgent queue */
            else
                wait.add( so_r ); /* add so_r into the wait queue */
            End if
        End if
    End if
End for
For each inter-session pattern p_inter , where p_inter ∈ P_inter
    If ( SO_S is contained in p_inter )
        {so_r} = the storage object set in p_inter requested right after SO_S ;
        For each so_r , where so_r ∈ {so_r}
            If ( so_r not in clients' Home Shark)
                calculate slack(so_r) ;
                If ( slack(so_r) ≤ 0))
                    urgent.add( so_r );/*add so_r into the urgent queue*/
                else
                    wait.add( so_r ); /* add so_r into the wait queue */
                End if
            End if
        End for
    End if
End for
```

Figure 4.6 : Procedure Predict_Related_Objects

Procedure Process_Urgent_Objects is triggered whenever the urgent queue is not empty. The procedure caches related storage objects in the urgent queue, $\{so_{r_u}\}$. Procedure Process_Wait_Objects is triggered if the wait queue is not empty and the network traffic between Sharks is below a user-defined threshold. The procedure caches related storage objects in the wait queue, $\{so_{r_w}\}$. Procedure Transfer_Objects transfers a related storage object in the wait queue, so_{r_w}, to the urgent queue whenever its wait expiration time, $wet(so_{r_w})$, has passed the current system time. Here, $wet(so_{r_w})$ is calculated as:

$$wet(so_{r_w}) = \text{time}(so_d) + slack(so_{r_w}) \tag{4.3}$$

where $\text{time}(so_d)$ is the request time for the storage object on demand, so_d.

4.5 The Network Storage Caching (NSC) Simulator

Simulation is a major tool to evaluate different caching approaches (Davison 2001). Simulations used in previous caching research, such as (Kroeger et al. 1997), and simulators developed for caching performance evaluation, such as NCS (Davison 2001) and PROXIM (Feldmann et al. 1999), are trace-driven simulations based on specific trace logs. Using trace-driven simulations, it is hard to isolate, examine and explain the impacts of such system characteristics as the size of cache space on the performance of different caching approaches. Unlike trace-driven simulators, the NSC simulator is based on general and proven characteristics of trace logs and provides researchers with flexibility

to manipulate parameters of these characteristics. Implemented using Csim18[19], the NSC simulator simulates how NetShark provides storage services for clients distributed in *m* geographically separated regions. At the beginning of a simulation run, storage objects are randomly allocated to *m* Sharks.

As shown in Figure 4.7, the NSC simulator simulates request generation and three types of servers: Client-Shark network, Shark and Shark-Shark network. There may be more than one server in each server type. These servers serve requests and deliveries of storage objects according to the First-Come-First-Served (FCFS) queuing discipline. In Section 4.5.1, we explain the request generation model and Section 4.5.2 describes the server queuing models.

[19] Csim18 is a process-oriented, discrete-event simulation tool developed by Mesquite Software, Inc.

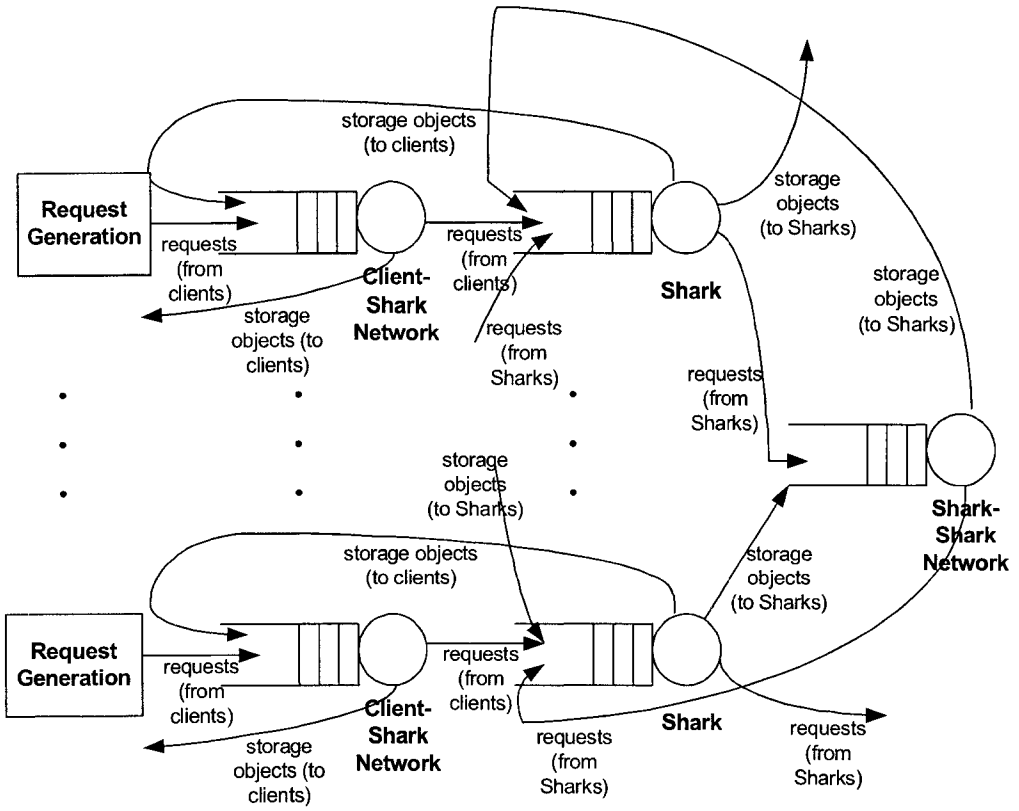


Figure 4.7: The NSC Simulator

4.5.1 The Request Generation Model

A request generation procedure consists of two steps: (1) generating session-begin messages and session-end messages; (2) generating requests within sessions. To generate session-begin/end messages, we assume the interarrival times between sessions and the durations of sessions to be exponentially distributed. Requests generation within a session begins as soon as the session-begin message has been generated and the requests generation stops as soon as the session-end message has been generated.

We call requests within a session as a request stream. We assume that the interarrival times between requests in a request stream follow a lognormal distribution (Paxson and Floyd 1995). As shown in Figure 4.8, a request stream is generated based on the request frequency rates of storage objects and the correlations between storage objects. Hence, the request frequency rates of storage objects and the correlations between storage objects need to be generated before a meaningful request stream can be generated.

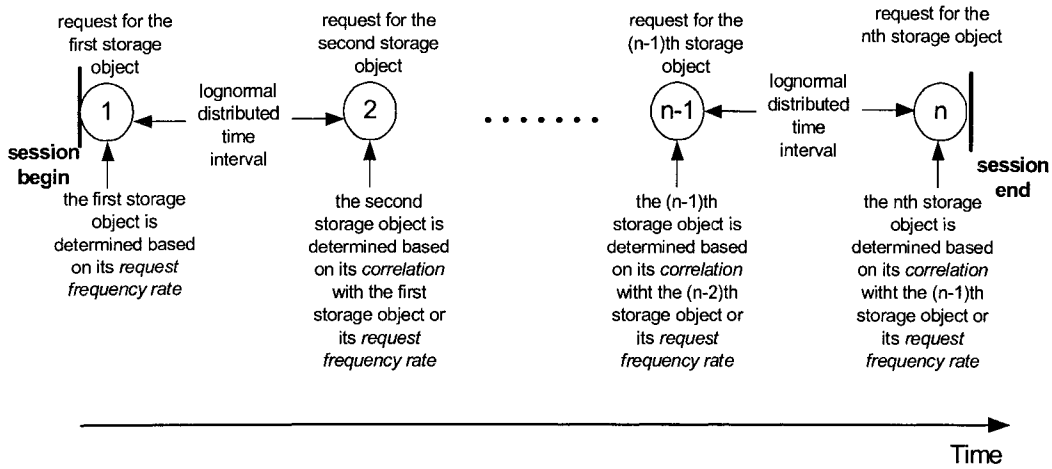


Figure 4.8: Request Stream Generation

The request frequency rates of storage objects approximately follow the Zipf's law in which the request frequency rate of the kth most frequently requested storage object is proportional to $\frac{1}{k}$ (Breslau et al. 1999). We denote the total number of storage objects in NetShark as n_{NS} and the set of all storage objects in NetShark as SO_{NS}, where $SO_{NS} = \{so_i\}$ for $i = 1,2,\cdots,n_{NS}$. For a storage object in NetShark, so_i, where $so_i \in SO_{NS}$ for $i = 1,2,\cdots,n_{NS}$, we denote $rank(so_i)$ as the rank of its request frequency

rate among all storage objects in NetShark. Let $rank(so_i) = k$ if so_i is the kth most frequently requested storage object in NetShark. According to (Breslau et al. 1999), the request frequency rate of so_i, $fre(so_i)$, is defined below.

$$fre(so_i) = \frac{c}{rank(so_i)} \qquad \text{where } c = \frac{1}{\sum_{j=1}^{n_{NS}} \frac{1}{j}} \tag{4.4}$$

c in (4.4) can be approximated using the following formula.

$$c \approx \frac{1}{\ln(n_{NS}) + C + \frac{1}{2n_{NS}}} \tag{4.5}$$

Here, C is the Euler constant and $C \approx 0.577$. Using (4.4) and (4.5), the request frequency rates of storage objects can be generated by the NSC simulator.

To simulate a situation in which some storage objects are often requested together within sessions, the NSC simulator generates correlations between storage objects. For a storage object in NetShark, so_i, where $so_i \in SO_{NS}$ for $i = 1,2,\cdots,n_{NS}$, we name a storage object requested right after so_i within the same session as the next requested storage object of so_i. There are two types of the next requested storage object of so_i:

- a correlated storage objects of so_i: requested frequently right after so_i;
- a randomly requested storage object of so_i: requested infrequently right after so_i.

For a storage object, so_i, where $so_i \in SO_{NS}$ for $i = 1,2,\cdots,n_{NS}$, we denote $fanout(so_i)$, namely the fan-out number of so_i, as the number of the correlated storage objects of so_i. $c_j(so_i)$ denotes a correlated storage object of so_i and $\{c_j(so_i)\}$ denotes the set of all correlated storage objects of so_i, where $c_j(so_i) \in SO_{NS}$ for $j = 1,2,\cdots, fanout(so_i)$. $P(c_j(so_i) \mid so_i)$ denotes the correlation between $c_j(so_i)$ and so_i, which is the conditional probability of requesting $c_j(so_i)$ right after requesting so_i within the same session. We denote $random(so_i)$, namely the random factor of so_i, as the conditional probability of requesting a randomly requested storage object of so_i right after requesting so_i, where,

$$random(so_i) + \sum_{j=1}^{fanout(so_i)} P(c_j(so_i) \mid so_i) = 1 \tag{4.6}$$

Example 4.5: As shown in Figure 4.9, storage object so_4 has two correlated storage objects, so_7 and so_9, i.e., $fanout(so_4) = 2$, $c_1(so_4) = so_7$ and $c_2(so_4) = so_9$. The correlation between so_7 and so_4, $P(so_7 \mid so_4)$, is 0.35 and the correlation between so_9 and so_4, $P(so_9 \mid so_4)$, is 0.55. The conditional probability of requesting a randomly requested storage object of so_4 right after requesting so_4, $random(so_4)$, is 0.10.

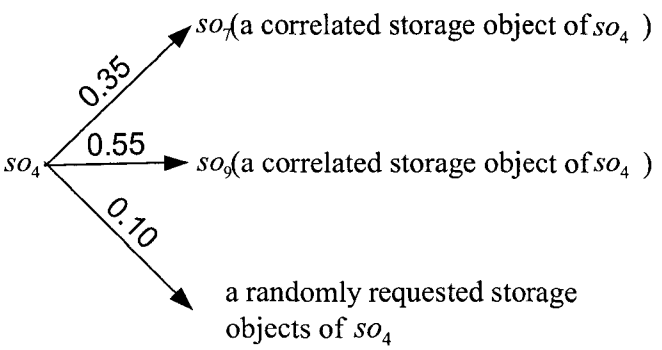


Figure 4.9: Correlations Between Storage Objects

Figure 4.10 gives an algorithm to generate all correlated storage objects of so_i, $\{c_j(so_i)\}$, and their correlations with so_i, $P(c_j(so_i) \mid so_i)$, where $c_j(so_i) \in SO_{NS}$ for $j = 1,2,\cdots, fanout(so_i)$, given the fan-out number of so_i, $fanout(so_i)$, and the random factor of so_i, $random(so_i)$. In the algorithm, $F(k)$ is denoted as the kth cumulative request frequency rate, where $k = 0,1,2,\cdots, n_{NS}$.

$$F(k) = \begin{cases} \sum_{m \le k} fre(so_m) & k = 1,2,\cdots, n_{NS},\ so_m \in SO_{NS} \text{ for } m = 1,2,\cdots, n_{NS} \\ 0 & k = 0 \end{cases} \tag{4.7}$$

In the NSC simulator, $fanout(so_i)$ is simulated using a Poisson random variable with parameter λ and $random(so_i)$ is simulated using a uniform random variable over the interval $(0, r)$, where $0 < r < 1$. λ and r can be manipulated to change correlations

between storage objects. Running the algorithm for every storage object in NetShark, correlations between storage objects in NetShark can be generated.

Input: $fanout(so_i)$: the fan-out number of so_i

$random(so_i)$: the random factor of so_i

Output: $\{c_j(so_i)\}$: the set of all correlated storage objects of so_i,

where $c_j(so_i) \in SO_{NS}$ for $j = 1,2,\cdots, fanout(so_i)$

$P(c_j(so_i) \mid so_i)$: the correlation between $c_j(so_i)$ and so_i,

where $c_j(so_i) \in SO_{NS}$ for $j = 1,2,\cdots, fanout(so_i)$

$corr = 1 - random(so_i)$; /* probability of requesting correlated storage objects of so_i */

$\{c_j(so_i)\} = \phi$;

While ($fanout(so_i) \geq 1$)

$x = rand(0,1)$; /* x ~ uniform distribution over (0,1)*/

/*pick a storage object based on its request frequency rate*/

select k, where $x \in [F(k-1), F(k))$;

If $so_k \in \{c_j(so_i)\}$ /* so_k has been selected before*/

continue;

End if

$\{c_j(so_i)\} = \{c_j(so_i)\} \cup \{so_k\}$;

If ($fanout(so_i) > 1$)

$P(so_k \mid so_i) = rand(0, corr)$;

$corr = corr - P(so_k \mid so_i)$;

else

$P(so_k \mid so_i) = corr$; /*last correlated storage object of so_i */

End if

$fanout(so_i)--$;

End while

Figure 4.10: Generate Correlated Storage Objects of A Storage Object

Figure 4.11 gives the request stream generation algorithm. In the algorithm, $F(t, so_i)$ denotes the tth cumulative correlation of so_i, where $t = 0,1,2,\cdots, fanout(so_i)$.

$$F(t, so_i) = \begin{cases} \sum_{j \le t} P(c_j(so_i) \mid so_i) & t = 1,2,\cdots, fanout(so_i) \\ 0 & t = 0 \end{cases} \tag{4.8}$$

Example 4.6: For correlations given in Example 4.5, we have,

$F(1, so_4) = P(c_1(so_4) \mid so_4) = P(so_7 \mid so_4)$=0.55;

$F(2, so_4) = P(c_1(so_4) \mid so_4) + P(c_2(so_4) \mid so_4) = P(so_7 \mid so_4) + P(so_9 \mid so_4)$ =0.9.

The request stream generation algorithm is triggered whenever there is a session-begin message. The first requested storage object in a request stream is generated based on its request frequency rate. The rest of the requested storage objects are generated using a loop that stops when a session-end message has been received. Within the loop,

- if a randomly generated number (i.e., uniformly distributed) is less than one minus the random factor (i.e., *random*(·)) of the currently requested storage object , a storage object is generated based on its correlation with the currently requested storage object;

- otherwise, a storage object is generated based on its request frequency rate.

Event: a session-begin message has been generated

Input: $fre(so_i)$: request frequency rates of storage objects,
where $so_i \in SO_{NS}$ for $i = 1,2,\cdots,n_{NS}$,
$P(c_j(so_i) \mid so_i)$: correlations between storage objects,
where $so_i \in SO_{NS}$ for $i = 1,2,\cdots,n_{NS}$ and
$c_j(so_i) \in SO_{NS}$ for $j = 1,2,\cdots, fanout(so_i)$.

Output: a request stream

/* generate the first requested storage object based on its request frequency rate*/
$x = rand(0,1)$; /* x ~ uniform distribution over (0,1)*/
select k, where $x \in [F(k-1), F(k))$;
place the request for so_k in the request stream;
current = so_k; /* currently requested storage object*/
While (session not end) /* not receiving a session-end message */
$x = rand(0,1)$;
If ($x < (1 - random(\text{current}))$)
/*generate a correlated storage object of the currently requested storage object*/
select t, where $x \in [F(t-1, \text{current}), F(t, \text{current}))$;
place the request for $c_t(current)$ in the request stream;
current = $c_t(current)$;
else
/* generate a randomly requested storage object based on its request frequency rate*/
$x = rand(0,1)$;
select k, where $x \in [F(k-1), F(k))$;
place the request for so_k in the request stream;
current = so_k;
End if
End while

Figure 4.11: The Request Stream Generation Algorithm

4.5.2 The Server Queuing Models

To simulate the servers, the sizes of storage objects need to be generated first. According to (Paxson and Floyd 1995), the sizes of storage objects follow a Pareto distribution, which is a type of heavy-tailed distribution with the following distribution function.

$$F(x) = 1 - (\frac{\alpha}{x})^{\beta} \qquad x \geq \alpha \tag{4.9}$$

Here α is the location parameter, which is the minimum size of all storage objects, and β is the shape parameter with the value ranging from 0.9 to 1.4 (Paxson and Floyd 1995). Using (4.9), the sizes of storage objects can be generated.

4.5.2.1 Client-Shark/Shark-Shark Network Servers

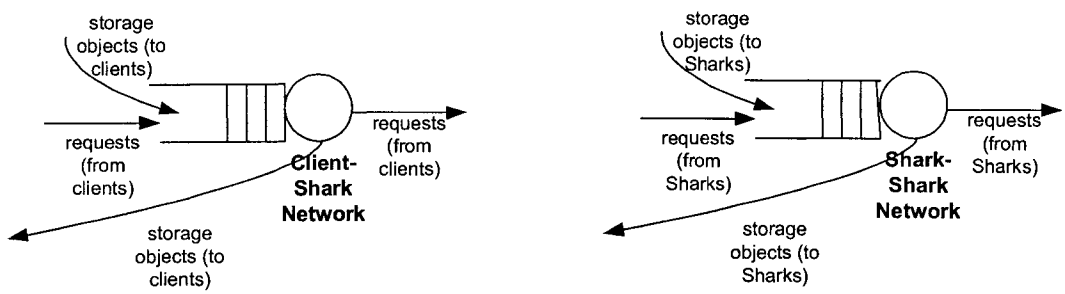


Figure 4.12: Client-Shark/Shark-Shark Network Servers

In the NSC simulator, Client-Shark network servers and Shark-Shark network servers are simulated using G/G/1 queues. As shown in Figure 4.12, both Client-Shark network servers and Shark-Shark network servers transfer requests and storage objects. The

distribution of the request interarrival times at a Client-Shark network server is determined by such factors as the distributions of the session and request interarrival times at its connected request generation model. The distributions of the request interarrival times at a Shark-Shark network server is determined by such factors as the hit rates of its connected Shark servers. The request service times of Client-Shark and Shark-Shark network servers can be estimated using (4.1). The distributions of the storage object interarrival times at Client-Shark and Shark-Shark network servers are determined by such factors as the waiting times and service times of their connected Shark servers. The storage object service times of Client-Shark and Shark-Shark network servers can be estimated using (4.1).

4.5.2.2 Shark Servers

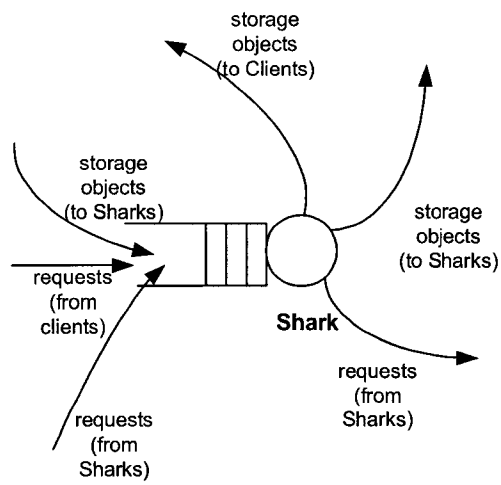


Figure 4.13: A Shark Server

As shown in Figure 4.13, a Shark server receives requests from clients or other Shark servers and retrieve storage objects to fulfill these requests. Requests that cannot be fulfilled and requests for prefetched storage objects are sent to other Shark servers. A Shark server also receives cached and prefetched storage objects from other Shark servers and store them. In the NSC simulator, a Shark server is simulated using a G/G/1 queue. The distributions of request and storage object interarrival times at a Shark server are determined by such factors as the distributions of request and storage object interarrival times at its connected Client-Shark/Shark-Shark network servers and request and storage object service times of these network servers. The service time of a Shark server consists of two parts: the computational time of the online caching algorithm running at the Shark server, which is considered as a constant in the NSC simulator; and the time to retrieve/store a storage object, *so*, from/to a Shark server, $IO(so)$, which is defined below.

$$IO(so) = \frac{size(so)}{dio} \tag{4.10}$$

Here *dio* is the disk I/O speed of Sharks.

4.6 Simulation Results

Using the NSC simulator described in Section 4.5, we compared three caching approaches: the data-mining-based prefetching approach described in this chapter, caching on demand and a widely used prefetching approach – the popularity-based prefetching appraoch (Dias et al. 1996, Markatos and Chronaki 1998).

Before presenting the simulation results, we list values of major simulation parameters in Table 4.6. In this Table, mean of session interarrival times and mean of request interarrival times are calculated from a FTP log. By running a network routing software – VisualRoute, we found that the number of hops between clients and servers within the same region ranged from 3 (e.g., clients and servers are in an organization's Intranet) to 9 and the number of hops between clients and servers in different regions ranged from 14 to 22 or even more (e.g., clients and servers are in different countries). In the architecture of NetShark, clients and their Home Sharks are in the same region while Sharks are distributed into different regions. Hence, we set the number of hops between clients and their Home Sharks at 6 (i.e., average of 3 and 9) and the number of hops between Sharks at 18 (i.e., average of 14 and 22). In the simulator, we assume that clients and Sharks are connected using Intranet/Internet and Sharks are connected using ATM PSDN (i.e., Public Switched Data Network).

Table 4.6: Values of Major Simulation Parameters

Parameter	Value
The simulation time	12 hours
The number of regions	6
The number of Sharks	6
The number of storage objects	20000
The location parameter of the Pareto distribution for storage object size (α)	30 K bytes
The shape parameter of the Pareto distribution for storage object size (β)	1.06
Mean of session interarrival times	80.9844 seconds
Mean of request interarrival times	5.9894 seconds
The random factors of storage objects ($random(\cdot)$)	uniform distributed over (0, 0.3)
Mean of the fan-out numbers of storage objects ($fanout(\cdot)$)	100
The request size	50 bytes
The bandwidth of Client-Shark network ($B(nw_{CS})$)	500K bps
The bandwidth of Shark-Shark network ($B(nw_{SS})$)	155M bps
Average hop delay of Client-Shark/ Shark-Shark network ($HD(\cdot)$)	0.015 second
The number of hops on Client-Shark network ($HN(nw_{CS})$)	6
The number of hops on Shark-Shark network($HN(nw_{SS})$)	18
The distance of Client-Shark network ($D(nw_{CS})$)	20 Km
The distance of Shark-Shark network ($D(nw_{SS})$)	1000 Km
The signal velocity of Client-Shark network ($v(nw_{CS})$)	231000 Km/second
The signal velocity of Shark-Shark network ($v(nw_{SS})$)	205000 Km/second
The disk I/O speed of Sharks (*dio*)	43.1M bytes/second
Maximum cache size	10% of the total storage object sizes

To compare the three caching approaches on the same benchmark, we used the same replacement approach, LRU, when comparing them. Figure 4.14 and Figure 4.15 illustrate hit rates (Measurement 2) and client perceived latencies (Measurement 1) of the three caching approaches as the cache size increased from 22% of the maximum cache size to 100% of the maximum cache size with an 11% gap. Compared with caching on demand, the popularity-based prefetching approach increased hit rate by an average of 21.8% and the data-mining-based prefetching approach increased hit rate by an average of 32.8%. Compared with caching on demand, the popularity-based prefetching approach reduced client perceived latency by an average of 5.1% and the data-mining-based prefetching approach reduced client perceived latency by an average of 7.7%.

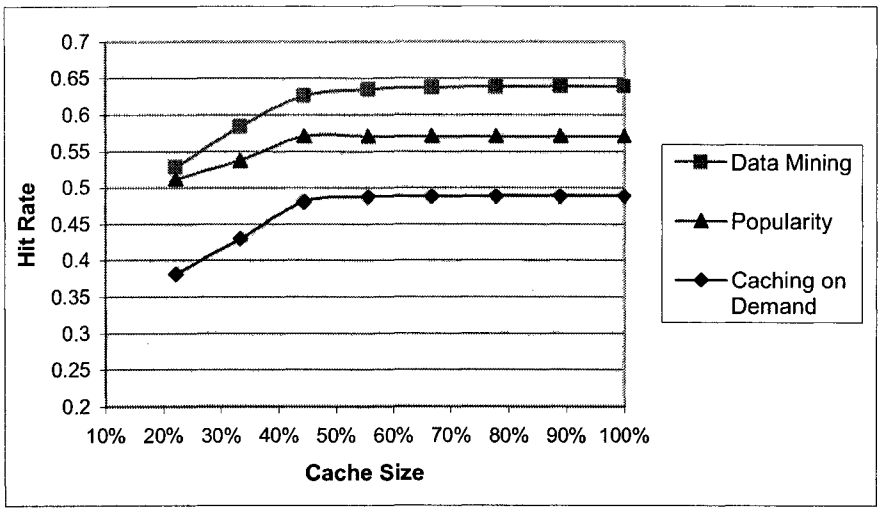


Figure 4.14: Hit Rate Comparison Between The Three Caching Approaches

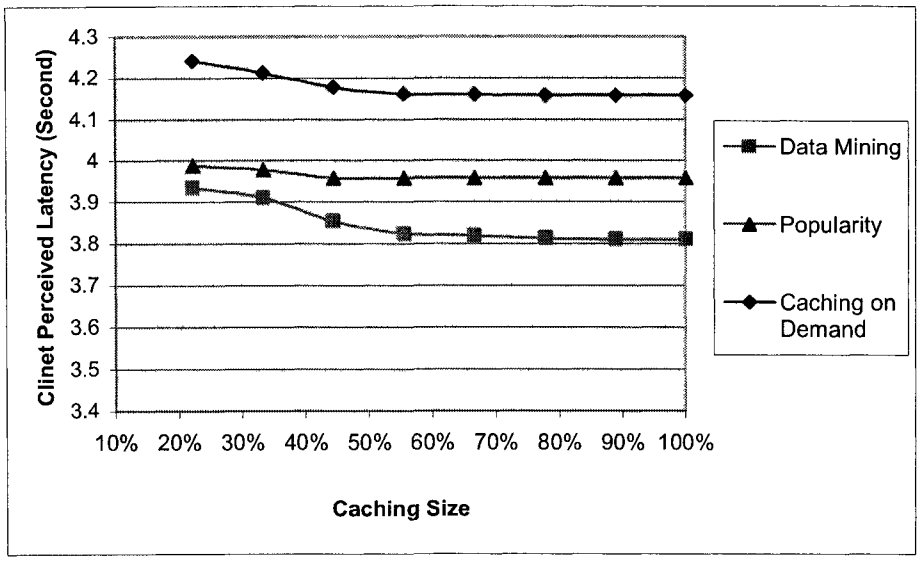


Figure 4.15: Client Perceived Latency Comparison Between The Three Caching Approaches

It is not surprising that both prefetching approaches outperformed caching on demand, as shown previously in (Kroeger et al. 1997). Between the two prefetching approaches, the data-mining-based prefetching approach outperformed the popularity-based prefetching approach for the following reasons. The popularity-based prefetching approach prefetches storage objects only based on their popularities (i.e., request frequencies) while the data-mining-based prefetching approach prefetches storage objects based on both their popularities (i.e., support of intra/inter-session patterns) and their correlations. Hence, the latter is more effective in prefetching storage objects that will be requested by clients. Furthermore, the data-mining-based prefetching approach considers the network traffic increase problem associated with prefetching approaches and tries to cache storage objects when the network traffic is not heavy. As shown in Figure 4.16, compared with caching on demand, the popularity-based prefetching approach increased Shark-Shark

response time (Measurement 3) by an average of 2.3% while the data-mining-based prefetching approach increased Shark-Shark response time by an average of 1.13%.

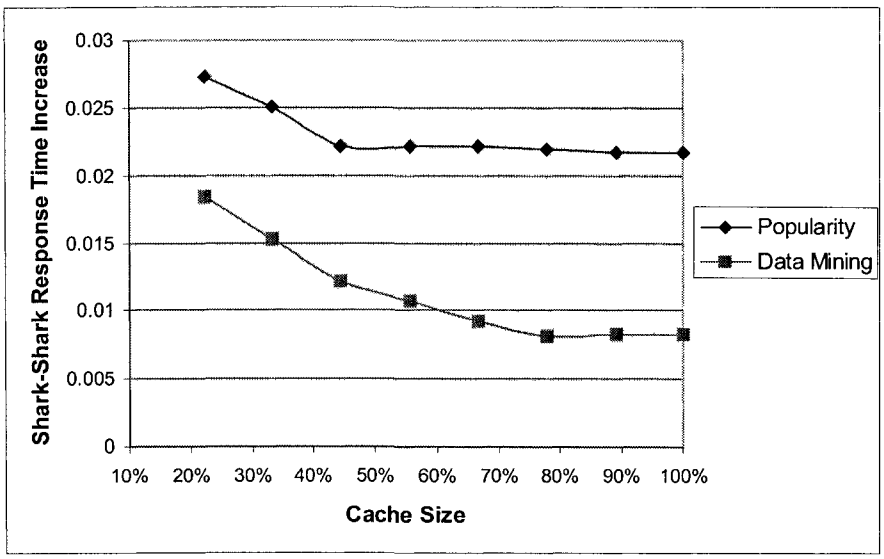


Figure 4.16: Network Traffic Increase Compared With Caching On Demand

4.6 Conclusion

Network storage is a key technique to solve the local storage shortage problem and to realize storage outsourcing over the Internet. This chapter has presented the implementation of a caching-based network storage system — NetShark and has proposed a caching approach – a data-mining-based prefetching approach. A simulator has been developed to evaluate the proposed caching approach. Simulation results have shown that the proposed caching approach outperforms other popularly used caching approaches.

As NetShark is used continuously, data in Log-DB accumulate continuously. Hence, intra/inter-session patterns learned from offline data mining need to be refreshed accordingly. This could also be categorized as a knowledge refreshing problem and be solved based on the model and optimal solutions developed in Chapter 2.

CHAPTER 5: IMPLEMENTATION OF THE OPTIMAL KNOWLEDGE REFRESHING POLICIES

In Chapter 2, we develop a Markov decision process model of knowledge refreshing and propose two optimal knowledge refreshing policies. We also study two real world KDD applications in Chapter 3 and Chapter 4 respectively. In this Chapter, we will discuss how to apply the model and the optimal knowledge refreshing policies developed in Chapter 2 to address knowledge refreshing requirements in the two real world KDD applications. Specially, the following research problems need to answered:

> Problem 1: Summarize real world knowledge refreshing requirements from the two KDD applications.
>
> Problem 2: Can the analytical model provide structural guidelines to analyze real world knowledge refreshing requirements? If so, how to implement the optimal knowledge refreshing policies?
>
> Problem 3: How to develop heuristics to address knowledge refreshing problems with changing arrival rates of queries and data.

In this chapter, we use Section 5.1 to study Problem 1 and Section 5.2 and Section 5.3 to address Problem 2. In Section 5.4, we propose heuristics for knowledge refreshing based on the optimal knowledge refreshing policies derived in Chapter 2.

5.1 Knowledge Refreshing Requirements for the Two KDD Applications

Figure 2.1 in Chapter 2 gives a general description of the knowledge refreshing problem. Similar to Figure 2.1, the following figure expresses the knowledge refreshing requirements for the web mining based web portal design problem, a KDD application described in Chapter 3.

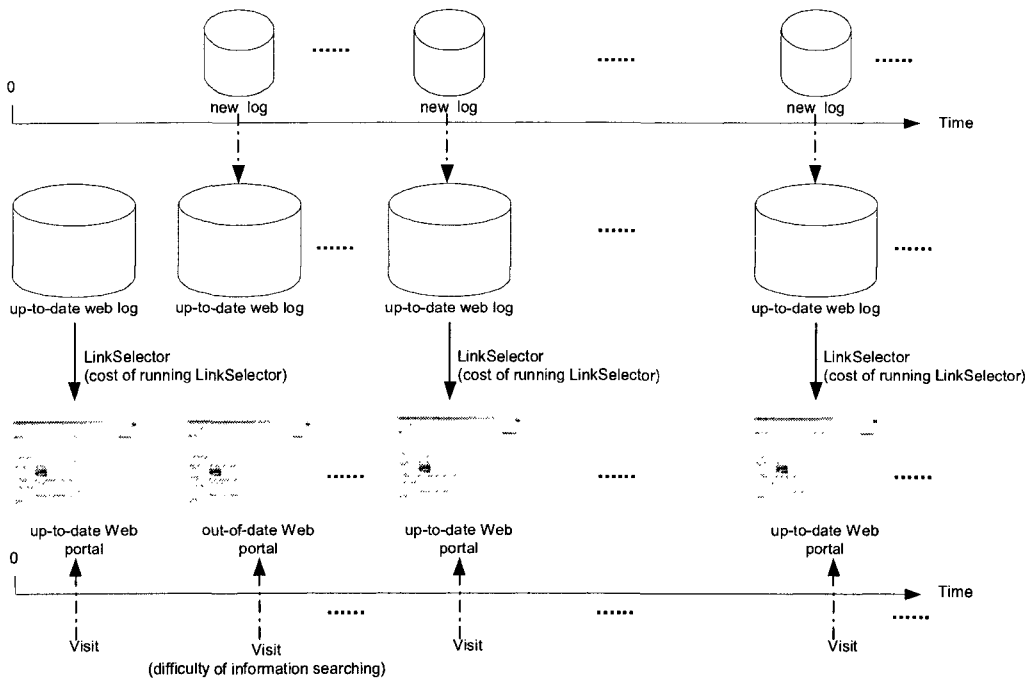


Figure 5.1: Knowledge Refreshing for Web Mining Based Web Portal Design

As described in Chapter 3, a web mining based web portal is generated when applying LinkSelector to a web log. In Figure 5.1, a web log evolves over time as new log continuously populated into it. New web log could bring in new web visiting patterns and

invalidate some old web visiting patterns discovered in a previous web mining. Changes in web visiting patterns will lead to changes in access relationships. As a result, a web portal generated by LinkSelector could be out-of-date. There are visits to the web portal to search for information. Out-of-date web portal could have a worse performance with respect to the three metrics of a web portal (see Section 3.3), which in turn increase the difficulty of information searching for visits to the web portal. To keep the generated web portal up-to-date with its underlying web log, LinkSelector has to be re-run, which incurs the cost of running LinkSelector. Hence, the knowledge refreshing problem in this KDD application is to find rules that can help determining time points over a time horizon to execute LinkSelector so that the tradeoff between costs of running LinkSelector and difficulties of information searching over the time horizon is optimized. We will discuss how to measure the difficulty of information searching and the cost of running LinkSelector in Section 5.2.

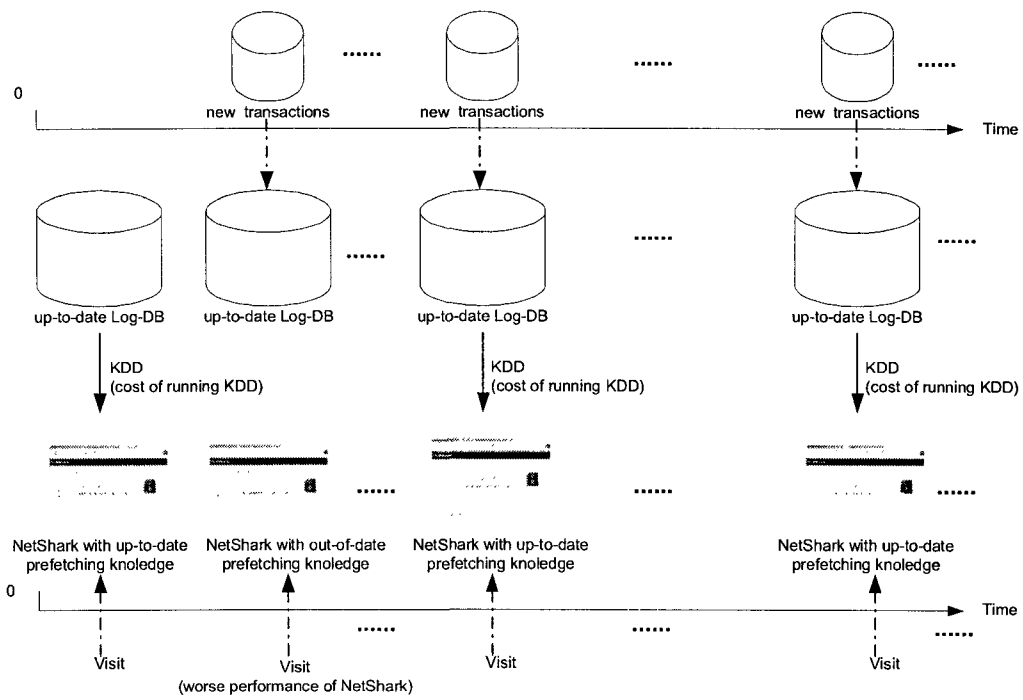


Figure 5.2: Knowledge Refreshing for Data Mining Based Caching

Similarly, Figure 5.2 describes the knowledge refreshing requirements for the data mining based caching approach, a KDD application described in Chapter 4. In Figure 5.2, Log-DB evolves over time as new transactions continuously populated into it. As a result, prefetching patterns (i.e., intra/inter-session patterns introduced in Section 4.4.1) discovered from Log-DB could also change over time. Out-of-date prefetching patterns will impact the data mining based caching approach negatively, which in turn impact the performance of the NetShark system negatively with respect to the three measurements described in Section 4.2. To keep the prefetching patterns up-to-date with Log-DB, a KDD run has to be executed, which incurs the cost of running KDD. Therefore, the

knowledge refreshing problem in this KDD application is to find rules that can help determine time points over a time horizon to execute KDD so that the tradeoff between costs of running KDD and the performance of the NetShark system over the time horizon is optimized.

In summary, knowledge refreshing requirements from both KDD applications can be described using the general structure of the knowledge refreshing problem introduced in Section 2.1 (see Figure 2.1). In Section 5.2, we explain how the analytical model proposed in Section 2.1 can capture real world knowledge refreshing requirements.

5.2 Parameter Estimation

The Markov decision process model proposed in Section 2.1 consists of the following components: decision point, action space, state space, transition probability, cost structure and objective function. The decision points for LinkSelector are the moments when a visit to the web portal arrives. At a decision point, running LinkSelector results in an up-to-date web portal while incurs a cost of running LinkSelector. On the other hand, not running LinkSelector saves a cost of running LinkSelector while makes a visit suffer from difficulty of information searching because of out-of-date web portal. Hence, the action space for LinkSelector can be defined as a binary set $A = \{0,1\}$, where 0 denotes not running LinkSelector and 1 denotes running LinkSelector.

For LinkSelector, the system state at decision point m is represented by a vector $s_m = (v_m^a, o_m)$, where v_m^a denotes the type of the visit arrives at decision point m and o_m denotes the degree of obsoleteness of the web portal at decision point m. The set of all feasible visits at decision point m is $\{v_i\}$, for $i = 1,2,\cdots,n$. Let us consider applying LinkSelector to design a Business-to-Consumer web portal. There could be several distinct types of visits to the web portal. For example, a lot of visits to the web portal just browse the web portal (i.e., visits of window shoppers); some visits could generate purchase in the near future (i.e., visits of potential buyers) while the rest really purchase during the visits (i.e., visits of buyers). Depending on real world situations, there are criteria to differentiate among window shoppers, potential buyers and buyers. For example, based on past customer web portal visiting data, simple criteria to differentiate among window shoppers, potential buyers and buyers could be: (1) customers whose ratio of purchasing visits out of total visits is above a threshold h_1 are buyers; (2) customers whose ratio of purchasing visits out of total visits is below a threshold h_2 are window shoppers; (3) customers whose ratio of purchasing visits out of total visits is between h_2 and h_1 are potential buyers. More complicated criteria could be developed using customer profiling techniques. After differentiating distinct types of visits, we can estimate their arrival rates, λ_{v_i}, for $i = 1,2,\cdots,n$, using past customer web portal visiting data.

Pattern deviation at decision point m, p_m, can be used as an approximate measurement of the degree of obsoleteness of the web portal at decision point m, o_m. Here, p_m is the difference between patterns discovered from a data source at decision point r_m, the latest decision point with action 1 before decision point m, and patterns discovered from a data source at decision point m. As LinkSelector is based on association rule mining, we define p_m for association rule mining as below,

$$p_m = 1 - \frac{|L_{r_m} \cap L_m|}{|L_{r_m} \cup L_m|} \tag{5.1}$$

In (5.1), L_{r_m} denotes large itemsets discovered at decision point r_m and L_m denotes large itemsets discovered at decision point m. It is clear from the definition of p_m that,

(1) $p_m = 0$ if and only if $L_{r_m} = L_m$, which indicates that the web portal generated at decision point r_m can still capture the up-to-date web visiting patterns at decision point m. Hence, at decision point m, the web portal generated at decision point r_m is still up-to-date;

(2) $p_m = 1$ if and only if L_{r_m} is totally different from L_m, which indicates that the web portal generated at decision point r_m can capture no web visiting pattern at decision point m. Hence, at decision point m, the web portal generated at decision point r_m is totally out-of-date;

(3) $0 < p_m < 1$ indicates that the web portal generated at decision point r_m can capture part of the web visiting patterns at decision point m . Hence, at decision point m, the web portal generated at decision point r_m is partially out-of-date.

From the above discussion, we can conclude that p_m indicates the degree of obsoleteness of the web portal at decision point m. The higher the p_m, the higher the degree of obsoleteness of the web portal at decision point m.

For LinkSelector, one step transition probability from state s_m to s_{m+1} under action a_m, $P^{a_m}_{s_m s_{m+1}}$, is $P\{v^a_{m+1} = v_i\}$, where $a_m \in$ A and $i = 1,2,\cdots,n$, because the transition form o_m to o_{m+1} is deterministic, given action a_m. At decision point m, the cost of running LinkSelector is a constant, denoted as C_L, because personnel costs dominate C_L. C_L equals to the number of person*days involved in running LinkSelector times daily salary.

At decision point m, the cost associated with the difficulty of information searching because of the obsoleteness of the web portal, c^o_m, is defined as,

$$c^o_m = p_m c_{v_i} \tag{5.2}$$

In (5.2), c_{v_i} is the cost of v_i, for $i = 1,2,\cdots,n$. For the Business-to-Consumer web portal example, cost of purchasing visit can be estimated as the average sales of previous purchasing visits to the web portal. Similarly, cost of potential purchasing visit can be

estimated as the average sales of previous potential purchasing visits to the web portal; while cost of window shopping visit can be estimated as the average sales of previous window shopping visits to the web portal. The system cost incurred at decision point m, $c(s_m, a_m)$, is,

$$c(s_m, a_m) = \begin{cases} c_m^o & \text{if } a_m = 0 \\ C_L & \text{if } a_m = 1 \end{cases} \tag{5.3}$$

The expected number of decision points over a relative long time horizon, M, is $T\sum_{j=1}^{n}\lambda_{v_j}$, where T is the duration of the time horizon. A policy of running LinkSelector over the time horizon π can be defined as $\pi = (\delta_1, \delta_2, \cdots, \delta_m, \cdots, \delta_M)$, where δ_m is the decision rule at decision point m such that $a_m = \delta_m(s_m)$. The total system cost over the time horizon under policy π, EC_π, can be expressed as,

$$EC_\pi = E\left\{\sum_{i=1}^{M}[\delta_i(s_i)C_L + (1-\delta_i(s_i))c_m^o]\right\} \tag{5.4}$$

According to the objective of knowledge refreshing for LinkSelector discussed in Section 5.1, the global optimal knowledge refreshing policy, π^*, minimizes total system cost over the time horizon such that $EC_{\pi^*} = \min_{\pi\in\Pi} EC_\pi$, where policy space Π is the set of all feasible policies.

In this section, we analyze a real world knowledge refreshing problem using the analytical model proposed in Section 2.1. The fit between the analytical model and the real world knowledge refreshing problem shows the applicability of the analytical model to real world knowledge refreshing problems. We also show how to estimate system parameters in the proposed Markov decision process model using the LinkSelector example. In the next section, an implementation procedure for knowledge refreshing will be discussed based on the optimal knowledge refreshing policies developed in Chapter 2.

5.3 Implementation Procedure for the Optimal Knowledge Refreshing Policies

As discussed in Chapter 2, the fixed optimal knowledge refreshing policy π_f^* is more computationally efficient and easier to implement that the global optimal knowledge refreshing policy π^*. However, the expected total system cost generated by implementing π^* is less than or equivalent to that generated by implementing π_f^*. In Lemma 3, two boundary conditions at which the expected total system cost generated by implementing π^* is equivalent to that generated by implementing π_f^* are derived. Hence, the criteria to choose which optimal knowledge refreshing policy to implement is:

(1) Choosing π_f^* to implement if one of the two boundary conditions in Lemma 3 is satisfied; or the saving of the expected total system cost generated by implementing π^* instead of π_f^* is with in an acceptable range.

(2) Otherwise, choosing π^* to implement.

Figure 5.3 gives a detailed implementation procedure for the optimal knowledge refreshing policies.

```
Estimate system parameters of the Markov decision process model;
Compute π*_f, EC_{π*_f} using Lemma 1;
If one of the conditions in Lemma 3 satisfied
        Implement π*_f as the knowledge refreshing policy;
Else
        Compute π*, EC_{π*} using dynamic programming;
        If (EC_{π*_f} − EC_{π*}) with in an acceptable range
                Implement π*_f as the knowledge refreshing policy;
        Else
                Implement π* as the knowledge refreshing policy;
        End if
End if
```

Figure 5.3: Implementation Procedure for the Optimal Knowledge Refreshing Policies

The implementation of π_f^* is trivial. It only needs to maintain an in-memory counter to record the number of queries at current decision point. The KDD is triggered to run if and only if the number of queries at current decision point divided by Q^* is an integer. The implementation of π^* requires to monitor system state over time. Two types of monitoring algorithms could be applied to monitor system state over time: the incremental monitoring algorithm (Fang and Liu Sheng 2000) and the data stream

monitoring algorithm (Zhu and Shasha 2002, Dong et al. 2003). The former returns exact system state while the latter returns approximate system state. However, only the latter is suitable for continuously monitoring system state over rapid and continuously populated data. Given the characteristics of current data intensive applications: continuously population of data and continuous queries to the knowledge base, the dissertation suggests using the data stream monitoring algorithm.

5.4 Heuristics for Knowledge Refreshing

As discussed in Section 2.2, π^* is the global optimal knowledge refreshing policy derived using the dynamic programming method. The advantage of π^* is the global optimality of the solution. However, the dynamic programming method is of limited utility when addressing real world problems (Sutton and Barto 1998), because of the following reasons.

(1) The dynamic programming method assumes a perfect model of the environment, which cannot hold for real world problems. For example, the dynamic programming method assumes a exact knowledge of transition probabilities and requires transition probabilities to be stationary over the time horizon. However, for the LinkSelector problem, only estimations of the arrival rates of web portal visits, λ_{v_i}, for $i = 1,2,\cdots,n$, can be obtained and λ_{v_i} could change over the time horizon.

(2) The dynamic programming method is computational expensive.

Therefore, to solve real world knowledge refreshing problems, heuristics need to be developed. This section describes a simple adaptive heuristic for knowledge refreshing – the limited look-ahead heuristic. In contrast to the dynamic programming method which generates decision rules for knowledge refreshing for the entire time horizon in advance, the limited look-ahead heuristic makes decisions only for the current decision point. Hence, the limited look-ahead heuristic has exact knowledge of the system state at each decision point. Consequently, the heuristic is adaptive in nature. At each decision point, the limited look-ahead heuristic makes decisions based on the short-term (i.e., current) system cost and the long-term system cost, in which the short-term system cost can be calculated exactly. The evaluation of the long-term system cost takes into account of a limited number of anticipated future system states and their corresponding actions. While using the dynamic programming method, the evaluation of the long-term system cost considers future system states and their corresponding actions from the current decision point until the end of the time horizon. As a result of the simplification, the limited look-ahead heuristic is much more computational efficient than the dynamic programming method. The tradeoff is that the limited look-ahead heuristic only generates sub-optimal knowledge refreshing policies.

We denote the number of look-ahead steps as LA, where $LA = 0,1,2,\cdots$. $LA = 0$ indicates that the heuristic only considers the short-term cost when making decisions. Let $h_{s_m}(m)$

be the minimum total system cost from decision point m to decision point (m+LA). $h_{s_m}(m)$ can be calculated as,

$$h_{s_m}(m) = \min_{a_m \in A}\{c(s_m, a_m) + \sum_{s_{m+1} \in S_{m+1}} P^{a_m}_{s_m s_{m+1}} h_{s_{m+1}}(m+1)\} \tag{5.5}$$

By calculating $h_{s_m}(m)$, we can also get the action to be taken at decision point m under the limited look-ahead heuristic. In (5.5), $h_{s_{m+1}}(m+1)$ can be calculated backward from decision point (m+LA) using the following formulas.

$$h_{s_{m+LA}}(m+LA) = \min_{a_m \in A}\{c(s_{m+LA}, a_{m+LA})\} \tag{5.6}$$

$$h_{s_k}(k) = \min_{a_m \in A}\{c(s_k, a_k) + \sum_{s_{k+1} \in S_{k+1}} P^{a_k}_{s_k s_{k+1}} h_{s_{k+1}}(k+1)\} \qquad \text{where } (m+1) \le k \le (m+LA-1) \tag{5.7}$$

When applying the limited look-ahead heuristic, (5.5) – (5.7) are calculated at each decision point. Actions that minimize (5.5) are actions suggested by the heuristic.

Both knowledge refreshing policies generated using the limited look-ahead heuristic and π_f^* are sub-optimal policies. It is important to know the performance comparison between the two policies. To examine this, the following experiments were performed under the simulation environment described in Section 2.3.1. To simulate the non-stationary arrival rates, we add an uniform random variable *noise*, distributed over $(-adj, adj)$, to λ_{q_1}, λ_{q_2}, λ_{q_3} and λ_d. The actual arrival rate of q_1 over the time horizon is

$\lambda_{q_1}(1+noise)$. Similarly, the actual arrival rates of q_2, q_3 and new data over the time horizon are $\lambda_{q_2}(1+noise)$, $\lambda_{q_3}(1+noise)$ and $\lambda_d(1+noise)$ respectively. All of the simulation results are collected from experiments with 500 simulation runs, yielding small confidence intervals for the simulation results.

Table 5.1 summarizes the results of the comparison between π_f^* and the knowledge refreshing policies generated using the 4-step look-ahead heuristic, $LA-4$, in which the length of the time horizon T is increased from 400 to 600 with a increment of 50 and the noise level is 0.4. In the table, performance improvement is the absolute difference between the total system cost under $LA-4$ and the total system cost under π_f^* divided by the latter, expressed in percentage.

Table 5.1: Performance Comparison Between π_f^* And $LA-4$ With Respect To The Duration of The Time Horizon ($adj = 0.4$)

T	$LA-4$	π_f^*	Performance Improvement (%)
400	1253.83	1636.84	23.39
450	1411.99	1852.05	23.76
500	1565.37	2066.78	24.26
550	1739.10	2253.83	22.83
600	1862.05	2446.21	23.88

We then set the duration of the time horizon to be 600 and the noise level to be 0.4. Table 5.2 summarizes the results of the comparison between π_f^* and the limited look-

ahead heuristic, in which the number of look-ahead steps is increased from 1 to 16 proportionally.

Table 5.2: Performance Comparison Between π_f^* And The Limited Look-Ahead Heuristic With Respect To The Number of Look-Ahead Steps ($T = 600, adj = 0.4$)

Number of look-ahead steps	LA	π_f^*	Performance Improvement (%)
1	2003.31	2446.21	18.10
2	1947.62	2446.21	20.38
4	1862.05	2446.21	23.88
8	1852.48	2446.21	24.27
16	1843.52	2446.21	24.63

We set the duration of the time horizon to be 600 and compare the performance between π_f^* and $LA-4$ with respect to the noise level. Table 5.3 summarizes the results of the comparison, in which the noise level is increased from 0.1 to 0.6 with a increment of 0.1.

Table 5.3: Performance Comparison Between π_f^* And $LA-4$ With Respect To The Noise Level ($T = 600$)

adj	$LA-4$	π_f^*	Performance Improvement (%)
0.1	1706.13	2181.88	21.80
0.2	1701.66	2183.49	22.06
0.3	1700.15	2187.51	22.27
0.4	1862.05	2446.21	23.88
0.5	1916.26	2547.66	24.78
0.6	1964.66	2641.78	25.63

All the experiment results have shown that the limited look-ahead heuristic outperforms π_f^*, as the limited look-ahead heuristic generates adaptive knowledge refreshing policies and more suitable for applications with evolving system parameters (e.g., non-stationary arrival rates). Specially, we can explicitly see the suitability of the limited look-ahead heuristic for applications with evolving system parameters from the experiment results in Table 5.3. In Table 5.3, the performance improvement of $LA-4$ increases as the noise level increases.

A critical problem associated with the limited look-ahead heuristic is how to determine the proper number of look-ahead steps. The following experiments shed some insight into the solution of the problem. We set the duration of the time horizon to be 600 and the noise level to be 0.3, 0.8 and 1.2 in three experiments respectively. Figure 5.4 to Figure 5.6 present the results of the experiments, in which the number of look-ahead steps is increased from 1 to 19 with a increment of 1step.

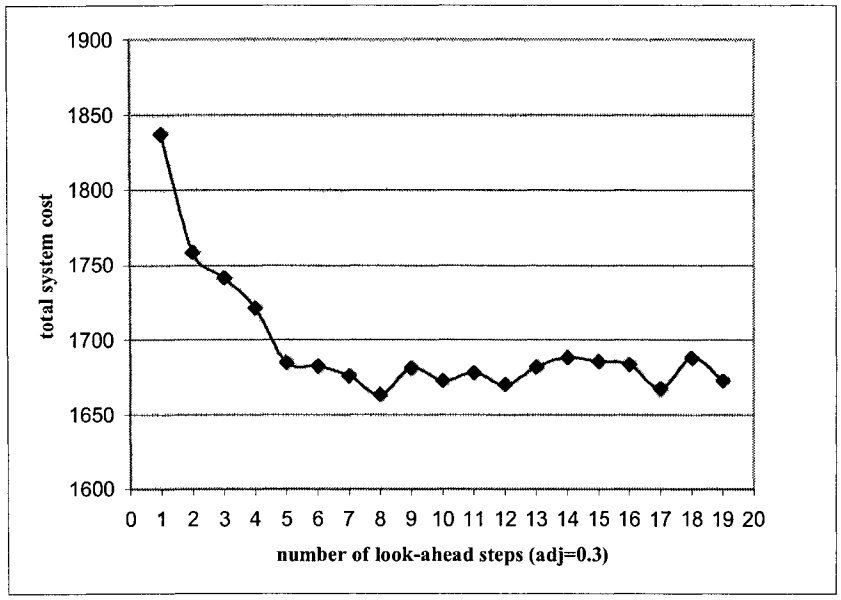


Figure 5.4: Total System Cost Vs. Number of Look-Ahead Steps (adj=0.3)

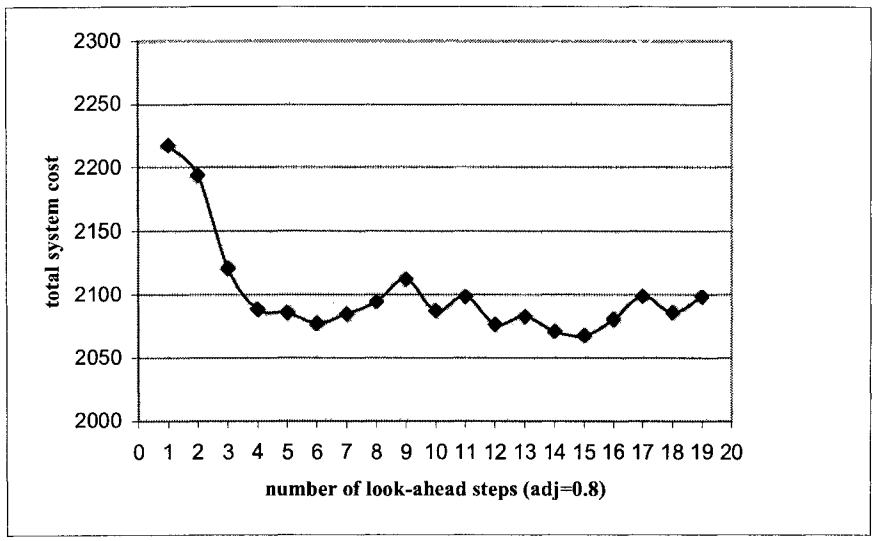


Figure 5.5: Total System Cost Vs. Number of Look-Ahead Steps (adj=0.8)

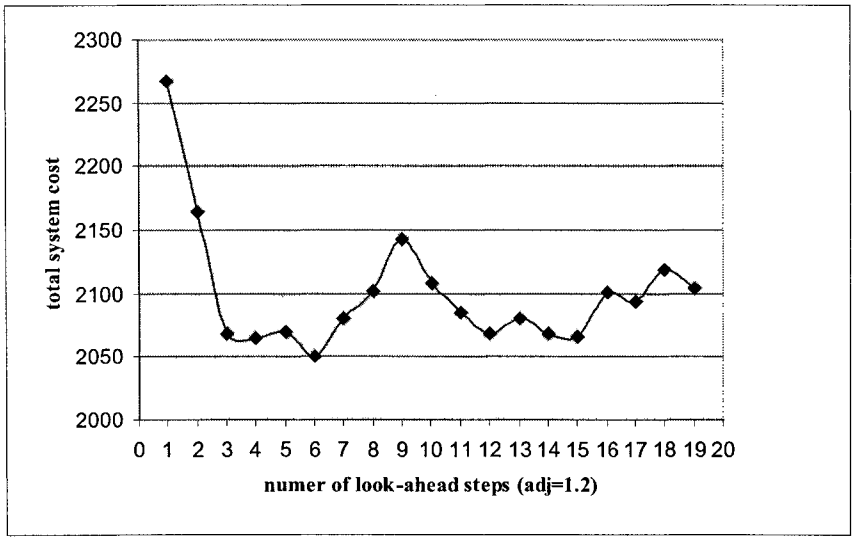


Figure 5.6: Total System Cost Vs. Number of Look-Ahead Steps (adj=1.2)

We summarize the observations from the three experiments as the following.

(1) When the noise level is set to be 0.3, the total system cost decreases as the number of look-ahead steps is increased from 1 to 8. Then, the total system cost fluctuates in a small range as the number of look-ahead steps is increased from 8 to 19.

(2) When the noise level is set to be 0.8, the total system cost decreases as the number of look-ahead steps is increased from 1 to 6. Then, the total system cost fluctuates in a middle range as the number of look-ahead steps is increased from 6 to 19.

(3) When the noise level is set to be 1.2, the total system cost decreases as the number of look-ahead steps is increased from 1 to 3. Then, the total system cost fluctuates in a large range as the number of look-ahead steps is increased from 3 to 19.

The observations indicate the following insights that can help us decide the number of look-ahead steps,

(1) There could be an upper bound for the number of look-ahead steps. Within the upper bound, increasing the number of look-ahead steps will benefit the total system cost. Beyond the upper bound, the benefit of increasing the number of look-ahead steps is uncertain;

(2) The upper bound decreases as the noise level increases.

The intuition behind the insights is that the impact of increasing the number of look-ahead steps are two-fold. Increasing the number of look-ahead steps could benefit the total system cost as it enables the decision making to consider further future situations. On the other hand, increasing the number of look-ahead steps could harm the total system cost as it brings in more uncertainty.

CHAPTER 6: CONCLUSION AND FUTURE WORKS

6.1 Conclusion and Contributions

The research problems addressed in this dissertation belong to two categories of research in the KDD area: applied research and fundamental research. For applied research, we research on how to apply KDD to such application areas as intelligent web portal design and web content management. For fundamental research, we identify a fundamental problem in the KDD area – knowledge refreshing, and propose an analytical model, solutions and heuristics for the problem. Figure 6.1 shows the relationship between the two categories of research. The fundamental research identifies and solves fundamental and general problems appearing in all KDD applications; while the applied research provides a test environment for solutions resulted from the fundamental research.

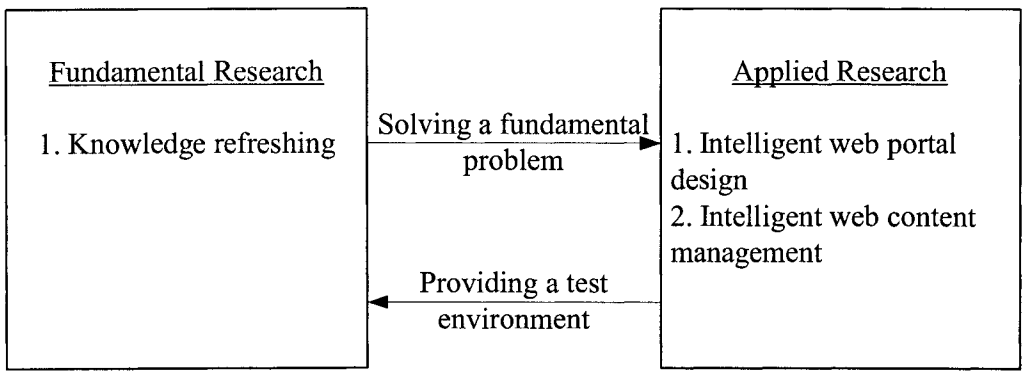


Figure 6.1: Relationship Between the Research Problems

In summary, the dissertation has made the following contributions. For the fundamental research,

(1) we have identified a fundamental problem in the KDD area – knowledge refreshing, and studied the problem from the distinct perspective of when to refresh knowledge;

(2) we have formally modeled the knowledge refreshing problem and derived solutions from the model;

(3) we have validated the model using the real world knowledge refreshing requirements and proposed implementation procedures and heuristics for knowledge refreshing.

The applied research addressed in this dissertation not only provides a test environment for the fundamental research problem – knowledge refreshing, but also has contributions in their respective application areas. For the research on intelligent web portal design,

(4) we have formally defined an important research problem in this area – hyperlink selection;

(5) we have proposed and shown that a web mining based hyperlink selection approach – LinkSelector, outperformed other hyperlink selection approaches.

For the research on web content management/web caching,

(6) we have proposed and shown that a data-mining-based prefetching approach outperformed other popularly applied web caching approaches;

(7) we have developed a web caching simulator based on general and proven characteristics of trace logs (e.g., FTP logs) to test the performance of different web caching approaches.

6.2 Future Works

Future works need to be done in both the fundamental research area and the applied research area. For the knowledge refreshing research, the following directions are suggested for future research.

(1) More analytical properties need to be derived from the Markov decision process model of knowledge refreshing. (a) In Lemma 3, we present a loose condition under which $EC_{\pi_f^*} = EC_{\pi^*}$. A more constrained condition under which $EC_{\pi_f^*} = EC_{\pi^*}$ needs to be developed. (b) We have explored the magnitude of cost saving between EC_{π^*} and $EC_{\pi_f^*}$ using numerical analysis. It would be highly interesting for decision makers to know (i) the condition under which the cost saving between EC_{π^*} and $EC_{\pi_f^*}$ is maximized; and (ii)

analytically, how the cost saving between EC_{π^*} and $EC_{\pi_f^*}$ varies with respect to the model parameters.

(2) All the proposed knowledge refreshing policies need to be tested and compared using real world data. Currently, they are tested and compared using simulation.

For the research on intelligent web portal design, future works include the following.

(3) We plan to conduct an empirical user study to examine the properties of different hyperlink selection approaches and compare their performances using the metrics proposed in Section 3.3 and other measurements in the usability study area.

(4) Current implementation of LinkSelector is based on access relationships mined from server web logs and structure relationships extracted from static web pages. Future research includes: how to collect cached web logs and combine them with server logs to mining access relationships and how to extract structure relationships from dynamic web pages.

Continued research on web content management/web caching consists of the following.

(5) We plan to incorporate structural relationships between storage objects (e.g., storage objects in the same directory) into the proposed caching approach.

(6) An analytical model based on the queuing theory needs to be developed to analyze the performance of different caching approaches mathematically.

REFERENCES

Aggarwal, C. C., P. S. Yu. 2001. A new approach to online generation of association rules. *IEEE Transactions on Knowledge and Data Engineering* 13(4) 527-540.

Agrawal, R., R. Srikant. 1994. Fast algorithms for mining association rules. *Proc. of the 20th VLDB Conference.*

Agrawal, R., T. Imielinski, A. Swami. 1993. Mining association rules between sets of items in large database. *Proc. of the ACM SIGMOD 1993 Conference.*

Agrawal, R., R., Srikant. 1995 Mining sequential patterns. *Proc. of the 1995 International Conference on Data Engineering (ICDE).*

Anderson, C., P. Domingos, D. Weld. 2001. Adaptive web navigation for wireless devices. *Proc. of the Seventeenth International Joint Conference on Artificial Intelligence.*

Armstrong, R., D. Freitag, T. Joachims, T. Mitchell. 1995. Webwatcher: a learning apprentice for the world wide web. *Proc. of the AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments.*

Babcock, B. et al. 2002. Models and issues in data stream systems. *Proc. of the ACM SIGMOD/PODS 2002 Conference.*

Baesebs, B., R. Setiono, C. Mues, J. Vanthienen. 2003. Using neural network rule extraction and decision tables for credit-risk evaluation. *Management Sci.* 49(3) 312-329.

Balabanovic, M., Y. Shoham. 1997. Fab: content-based collaborative recommendation. *Communications of the ACM*, 40(3): 66-72.

Barbara, D., et al. 1997. The New Jersey data reduction report. *IEEE Data Engineering Bulletin.* 20(4) 3-45.

Barish, G., K. Obraczka. 2000. World wide web caching: trends and techniques. *IEEE Communications Magazine*, May 2000, 178-185.

Bellman, R. E. 1957. Dynamic programming. *Princeton University Press, Princeton, NJ.*

Bourgeois, L. J., K. M. Eisenhardt. 1988. Strategic decision processes in high velocity environments: four cases in the microcomputer industry. *Management Sci.* 34(7) 816-835.

Breslau, L., P. Cao, L. Fan, G. Phillips, S. Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. *Proc. of IEEE INFOCOMM.*

Brin, S., R. Motwani, J. D. Ullman, S. Tsur. 1997. Dynamic itemset counting and implication rules for market basket data. *Proc. of the 1997 ACM SIGMOD,*255-264.

Brin, S., L. Page. 1998. The anatomy of a large-scale hypertextual web search engines. *Proc. of the Seventh International World Wide Web Conference.*

Can, F. 1993. Incremental clustering for dynamic information processing. *ACM Transactions on Information Systems.* 11(2) 143-164.

Cao, P., S. Irani. 1997. Cost-aware www proxy caching algorithm. *Proc. USENIX Symp. on Internet Technologies and Systems.*

Cao, P., C. Liu. 1998. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers* 47(4) 445-457.

Catledge, L., J. Pitkow. 1995. Characterizing browsing behaviors on the world wide web. *Computer Networks and ISDN Systems*, 27(6): 1065-1073.

Chang, C., M. Chen. 2002. Web cache replacement by integrating caching and prefetching. *Proceedings of the ACM International Conference on Information and* Knowledge Management.

Chakrabarti, J. et al. 1999. Mining the web's link structure. IEEE Computer, 32(8): 60-67.

Chakrabarti, J. 2000. Data mining for hypertext: a tutorial survey. *ACM SIGKDD Explorations*, 1(2): 1-11.

Chandy, K. M., J.C. Browne, C. W. Dissly, W. R. Uhrig. 1975. Analytical models for rollback and recovery strategies in data base systems. *IEEE Transactions on Software Engineering.* 1(1) 100-110.

Chen, M., J. Park, P. Yu. 1996. Data mining for path traversal patterns in a web environment. *Proc. 16th Intl. Conf. on Distributed Computing Systems.*

Cheung, D. W., J. Han, V. T. Ng, C. Y. Wong 1996. Maintenance of discovered association rules in large databases: an incremental updating technique. *Proc. of the 1996 Int'l Conf. on Data Engineering.*

Chinen, K., S. Yamaguchi. 1997. An interactive prefetching proxy server for improvement of www latency. *Proceedings of INET'97.*

Cooley, R., B. Mobasher, J. Srivastava. 1997. Web mining: information and pattern discovery on the world wide web. *IEEE International Conference on Tools with AI.*

Cooley, R., P. Tan, J. Srivastava. 1999a. WebSIFT: the website information filter system. *Proc. of the Web Usage Analysis and User Profiling Workshop.*

Cooley, R., B. Mobasher, J. Srivastava. 1999. Data preparation for mining world wide web browsing patterns. *Knowledge and Information Systems* 1(1) 1-27.

Crovella, M., P. Barford. 1998. The network effects of prefetching. *Proceedings of IEEE INFOCOM.*

Cooper, L. G., G. Giuffrida. 2000. Turning data mining into a management science tool: new algorithms and empirical results. *Management Sci.* 46(2) 249-264.

Dally, W. J. 2001. *Interconnection Networks.* A textbook in preparation can be accessed at http://cva.stanford.edu/ee482b/handouts.html

Davison, B. D. 2001. NCS: network and cache simulator – an introduction. *Technical Report DCS-TR-444,* Department of Computer Science, Rutgers University.

Davison, B. D. 2002. Predicting web actions from html content. *Proceedings of ACM 2002 Hypertext Conference.*

Dias, G. V., G. Cope, R. Wijayaratne. 1996. A smart internet caching system, *Proceedings of INET' 96 (The Internet Summit).*

Domingod, P., G. Hulten. 2000. Mining high-speed data streams. *Proc. of the 2000 ACM SIGKDD.*

Dong, G. et al.. 2003. Online mining of changes from data streams: research problems and preliminary results. *Proc. of the ACM SIGMOD MPDS 2003 Conference.*

Eisenhardt, K. M. 1989. Making fast strategic decisions in high-velocity environments. *Academy of Management Journal.* 32(3) 543-576.

Fayyad, U., G. Piatetsky-Shapiro, P. Smyth. 1996. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM.* 39(11) 27-34.

Fan, L., Q. Jacobson, P. Cao, W. Lin. 1999. Web prefetching between low-bandwidth clients and proxies: potential and performance. *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99).*

Fang, X., O. R. Liu Sheng. 2000. A monitoring algorithm for incremental association rule mining. *Proceedings of the 10th Workshop on Information Technology and Systems (WITS 2000), Brisbane, Australia.*

Feldmann, A., R. Caceres, F. Douglis, G. Glass, M. Rabinovich. 1999. Performance of web proxy caching in heterogeneous bandwidth environments. *Proceedings of IEEE INFOCOM.*

Ganti, V., J. Gehrke, R. Ramakrishnan, W. Y. Loh. 1999. A framework for measuring changes in data characteristics. *Proc. of the 18th ACM SIGMOD Symposium on Principles of Database Systems,* 126-137.

Ganti, V., J. Gehrke, R. Ramakrishnan. 2001. DEMON: mining and monitoring evolving data. *IEEE Transactions on Knowledge and Data Engineering* 13(1): 50-63.

Gibson, G., R. Meter. 2000. Network attached storage architecture. *Communications of The ACM* 43(11) 37-45.

Gwertzman, J., M. Seltzer. 1995. The case for geographically push-caching. *Proceedings of the 1995 Workshop on Hot Operating Systems*, Orcas Island, WA, May 1995.

Guha, S., N. Mishra, R. Motwani, L. O'Callaghan. 2000. Clustering data streams. *Proc. of the Annual Symposium. on Foundations of Computer Science (FOCS 2000)*

Guha, S., A. Meyerson, N. Mishra, R. Motwani, L. O'Callaghan. 2003. Clustering data streams: theory and practice. *IEEE Transactions on Knowledge and Data Engineering* 15(3): 515-528.

Hidber, C. 1999. Online association rule mining. *Proc. of the 1999 ACM SIGMOD.*

Hoeffding, W. 1963. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Assiociations.* 58 13-30.

Horng, Y., W. Lin, H. Mei. 1998. Hybrid prefetching for WWW proxy servers. *Proceedings of the 1998 IEEE International Conference on Parallel and Distributed Systems.*

Hulten, G. et al. 2001. Mining time-changing data streams. *Proc. of the 2001 ACM SIGKDD.*

Jin, S., A. Bestavros. 2000. Popularity-aware greedy dual-size web proxy caching algorithms. *Proceedings of ICDCS'2000*

Kim, S., J. Kim, J. Hong. 2000. A statistical, batch, proxy-side web prefetching scheme for efficient internet bandwidth usage. *Proceedings of the 2000 Networld+Interop Engineers Conference.*

Kleinberg, J., C. Papadimitriou, P. Raghavan. 1998. A microeconomic view of data mining. *Journal of Data Mining and Knowledge Discovery.* 2 311-324.

Kleinberg J. 1998. Authoritative sources in a hyperlinked environment. *Proc. of ACM-SIAM Symposium on Discrete Algorithms.*

Kroeger, T.M., D.D.E. Long, J.C. Mogul. 1997. Exploring the bounds of web latency reduction from caching and prefetching. *Proc. USENIX Symp. on Internet Technologies and Systems.* 13-22.

Kosala, R., H. Blockeel. 2000. Web mining research: a survey. *ACM SIGKDD Explorations*, 2(1): 1-15.

Lang, K. 1995. NewsWeeder: learning to filter netnews. *Proc. of the 12th International Conference on Machine Learning.*

Lawrence, S., C. L. Giles. 1998. Searching the world wide web. *Science* 280: 98-100.

Lawrence, S., C. L. Giles. 1999. Accessibility of information on the web. *Nature* 400: 107-109.

Lee, E. K., C. A. Thekkath. 1996. Petal: distributed virtual disks. *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 84-92.

Lee, J., M. Podlaseck. 2001. Visualization and analysis of clickstream data of online stores for understanding web merchandising. *Data Mining and Knowledge Discovery* 5(1-2): 59-84.

Li, Y. 1998. Towards A qualitative search engine. *IEEE Internet Computing*, 2(4): 24-29.

Lieberman, H. 1995. Letizia: an agent that assists web browsing. *Proc. of the 1995 International Joint Conference on Artificial Intelligence.*

Liu Sheng, O. R. 1992. Analysis of optimal file migration policies in distributed computer systems. *Management Science* 38(4) 459 - 482.

Lou, W., H. Lu. 2002. Efficient prediction of web access on a proxy server. *Proceedings of the ACM International Conference on Information and Knowledge* Management.

Manber, U., A. Patel, J. Robison. 2000. Experience with personalization on Yahoo!. *Communications of the ACM*, 43(8): 35-39.

Manku, G. S., R. Motwani. 2002. Approximate frequency counts over data streams. *Proc. of the 28th VLDB Conference.*

Markatos, E., C. E. Chronaki. 1998. A top-10 approach to prefetching on the web. *Proceedings of INET' 98 (The Internet Summit).*

Mladenic, D. 1999. Machine learning used by personal webwatcher *Proc. of ACAI-99 Workshop on Machine Learning and Intelligent Agents.*

Mobasher, B., R. Cooley, J. Srivastava. 2001. Automatic personalization based on web usage mining. *Communications of the ACM* 43(8): 142-151.

Molloy, M. K. 1989. Fundamentals of performance modeling. *Macmillan publishing company, New York.*

Nielson, J. 1999. User interface directions for the web. *Communications of the ACM* 42(1): 65-72.

Nielson, J., A. Wagner. 1996. User interface design for the WWW. *Proc. of ACM CHI 96.*

O'Callaghan, L. et al. 2002. High-performance clustering of streams and large data sets *Proc. of the 2002 Intl. Conference on Data Engineering.*

Padmanabhan, V. N., J. C. Mogul. 1996. Using predictive prefetching to improve world wide web latency. *Proceedings of the SIGCOMM'96 Conference.*

Park, J. S. et al. 1990. Optimal reorganization policies for stationary and evolutionary databases. *Management Sci.* 36(5) 613-631.

Paxson, V., S. Floyd. 1995. Wide-area traffic: the failure of poisson modeling, *IEEE/ACM Transactions on Networking* 3(3) 226-244.

Perkowitz, M., O. Etzioni. 2000. Towards adaptive websites: conceptual framework and case study. *Artificial Intelligence,* 118(1-2): 245-275.

Ross, S. M. 2000. Introduction to probability models. *Harcourt Academic Press.*

Quinlan, J.R. 1986. Induction of decision trees. *Machine Learning* 1 81-106.

Rastogi, R. 2003. Guest editor introduction: special section on online analysis and querying of continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15(3): 513-514.

Resnick, P. et al. 1994. GroupLens: An open architecture for collaborative filtering of netnews. *Proc. of the ACM Conference on CSCW.*

Salton, G. 1968. Automatic information organization and retrieval. *McGraw-Hill Inc.*

Sarkar, S., R. S. Sriram. 2001. Bayesian models for early warning of bank failures. *Management Sci.* 47(11) 1457-1475.

Schlimmer, J.C., D. Fisher. 1986. A case study of incremental concept induction. *Proc. of the 5th National Conference on Artificial Intelligence.*

Shardanand, U., P. Maes. 1995 Social information filtering: algorithms for automating "word of mouth" *Proceedings of ACM CHI'95.*

Sliberschatz, A. A. Tuzhilin. 1996. What makes patterns interesting in knowledge discovery systems. *IEEE Transactions on Knowledge and Data Engineering*. 8(6) 970-974.

Spiliopoulou, M., C. Pohle. 2001. Data mining to measure and improve the success of websites. *Data Mining and Knowledge Discovery* 5(1-2): 85-114.

Srivastava, J., R. Cooley, M. Deshpande, P. Tan. 2000. Web usage mining: discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1(2), 1-12.

Steinke, S. 2001. Network delay and signal propagation. *Network Magazine*. http://www.networkmagazine.com/article/NMG20010416S0006.

Sutton, R.S., A. G. Barton. 1998. Reinforcement learning: an introduction. *MIT Press*.

Tam, K. Y., M. Y. Kiang. 1992. Managerial application of neural network: the case of bank failure predictions. *Management Sci.* 38(7) 926-947.

Tanenbaum, A. S., A. S. Woodhull. 1997. Operating systems: design and implementation. *Prentice-Hall Inc.*

Thekkath, C. A., T. Mann, E.K. Lee. Frangipani: a scalable distributed file system. 1997. *Proceedings of the 16th ACM symposium on Operating Systems Principles.*

Thomas, S., S. Bodagala, K. Alsabti, S. Ranka. 1997. An efficient algorithm for the incremental updation of association rules in large databases. *Proc. of the 1997 ACM SIGKDD.*

Treharne, J. T., C. R. Sox. 2002. Adaptive inventory control for nonstationary demand and partial information. *Management Sci.* 48(5) 607-624.

Toivonen, H. 1996. Sampling large databases for association rules. *Proc. of the 22nd VLDB Conference.*

Utgoff, P.E. 1988. ID5: an incremental ID3. *Proc. of the 5th International Conference on Machine Learning.*

Utgoff, P.E. 1989. Incremental induction of decision trees. *Machine Learning* 4 161-186.

Yan, T., M. Jacobsen, H. Garcia-Molina, U. Dayal. 1996. From user access patterns to dynamic hypertext linking. *Proc. of the 5th International World Wide Web Conference.*

Yin, J., L. Alvisi, M. Dahlin, A. Iyengar. 2002. Engineering web caching consistency. *ACM Transactions on Internet Technology* 2(3) 224-259.

Young, J. W. 1974. A first order approximation to the optimum checkpoint interval. *Communications of ACM.* 17(9) 530-531.

Zhang, T., R. Ramakrishnan, M. Livny. 1996. BIRCH: an efficient data clustering method for very large databases. *Proc. of the 1996 ACM SIGMOD.*

Zhu, Y., D. Shasha. 2002. StatStream: statistical monitoring of thousands of data streams in real time. *Proceedings of the 28th VLDB Conference.*