DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS

INDEPENDENT STUDY REPORT

SUMMARY OF WORK DONE AS A SUMMER INTERN AT INKTOMI CORPORATION

Duration: May 22, 2000 to Aug 18, 2000

Prepared by: Ritesh Jain

To: Prof Geneva Belford

A short company profile

Inktomi is one of the fast developing companies in the Bay area. As opposed to the general view that Inktomi just develops search engines, its 60% work is in developing network products like Traffic Servers (to enhance access to the internet by caching the data locally). Of the remaining, 30% work is on search engines and the shopping engine takes the rest of the 10%.

I was working in the shopping engine and my group was the Targeting and Personalization group. This division of Inktomi is comparatively new and so people are very enthusiastic about working in this group. Their first major aim is to recover all the revenue spent over this group, and therefore I met some of the most talented and brilliant software engineers working in the Bay area in this division.

The main aim of our group was to personalize the interests of an Internet user by storing all the information available about that user and then targeting that user by showing him/her deals that could most interest him/her. It is a big project involving many smaller modules. I worked on three of these modules (one of which was an independent project given to me). They are listed below:

1. PullPage
2. External Profile Import (worked on this all alone)
3. ProfileCaching on TargetServer

The details of all of these tasks follow.

1. Introduction

Inktomi plays host or back-end to many merchants who want to use Inktomi's services and databases to display their products at their portals' websites. To store information about a particular merchant's products in Inktomi's database, a software called **PullPage** is written.

PullPage.exe is an independent software module to parse merchants' product page, identify product image, and remove all the links and unnecessary text and image. Whenever the Impulse Selling System at Inktomi sends UPDATE SERVER string commands through Microsoft Internet Information Server, UPDATE SERVER starts a separate process to invoke PullPage.exe for the purpose of pulling the product page, i.e., pulling out the product image and its description.

This project is to develop or maintain merchant filter to pull product pages from merchant web server. The merchant filter is inherited from a base class **merchantHtmlScriptObj**. The class **merchantHtmlScriptObj** has all the general methods to parse html-based product pages. What the merchant filter does is to identify product image and remove unnecessary texts and images based on common features existing in all the product pages of this merchant.

2. Development procedures and my contribution

1. Filter request

   I used to get a pullpage filter request with three sample product URLs whenever a new merchant joined Inktomi Merchandising Network, or there was an error in product image or pulled product page of existing merchant on the rack of live product offers.

2. Filter Modification or Implementation

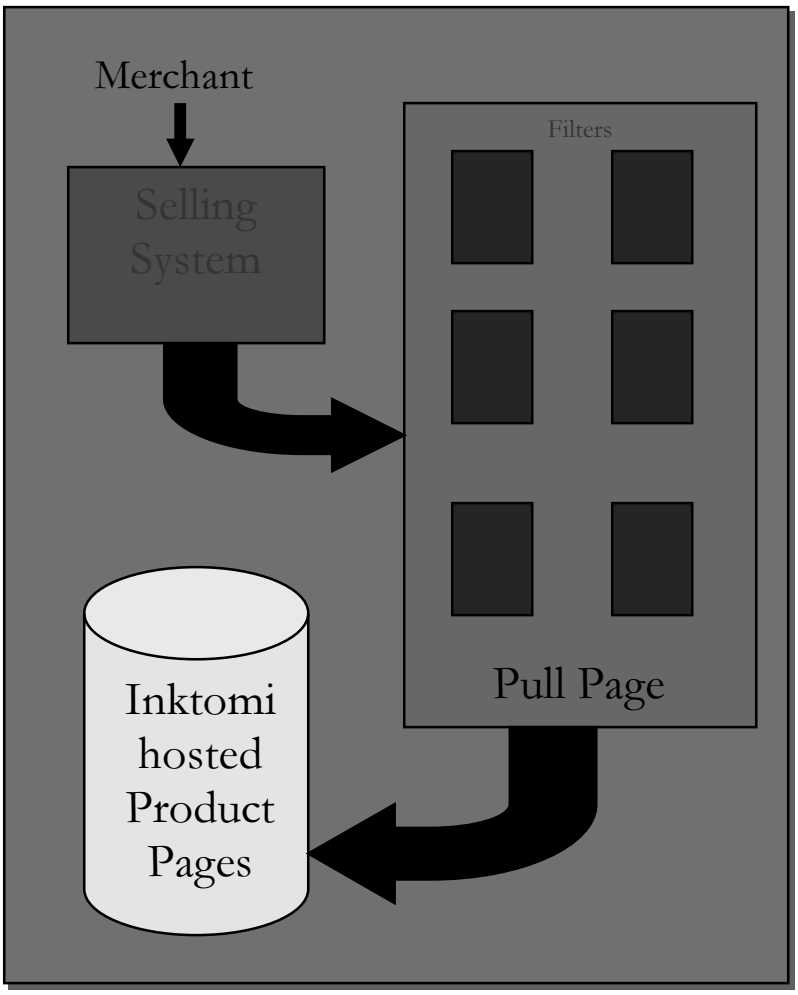
   My task was to generate a new merchant filter class or modify an existing merchant filter class. What I needed to do was to check the source code where this merchant filter was implemented to fix the problem in the existing merchant filter, or to derive a new merchant filter class from merchantHtmlScriptObj and then register it inside MerchantFilter.cpp. Since the layout of product pages of merchants can be very different, there could be no definite solution to some particular problems from different merchants.

   To create a new filter, you simply take the source code of the html page of the merchant product and identify the entire set of html tags used in the page along with their attributes and values. Then, parse those tags sequentially, ignoring all the unnecessary text and images and storing the product image and its description in a new html file, which is then stored in the Inktomi's database.

3. Some of the merchant filters that I developed:

StyleClick.com	NextGeneration.com	BreatheFree.com
Craftopia.com	Guild.com	CameraSphere.com

A block diagram of the whole process:



1. General Description

1.1. Overview

The aim was to load the user profile data i.e., a user's name, age, sex, location, interests etc., from the portal's database to the Inktomi database.

2. Functional Requirements

2.1. New language – UPML (User Profile Markup Language) (based on XML).

A new markup language using XML style was designed for the purpose. It has the following structure:

```
<UPML>
    <USER>
        <AFFILID ID = "3"/>
        <AFFILUSERID ID = "GO10"/>
        <OVERWRITE VALUE = "Y"/>
        <PROFILEVAR ID = "1002" PROB = "93"/>
        <PROFILEVAR ID = "1001" PROB = "1"/>
        <PROFILEVAR ID = "1003" PROB = "1"/>
        .
        .
        .
    </USER>
    .
    .
    .
</UPML>
```

So, basically, it contains information about each user like their affiliate id (or portal id), affiliateuser id, i.e., the id of the user for that affiliate, and a series of pair <demogid, value>. Demogid is nothing but a profile variable like 'age between 13 and 17', or 'owns a house' etc. and value is the probability of that user having that demogid. The overwrite flag, if 'Y' indicates that the previous data of the user (if any) in the Inktomi database allowed to be overwritten by the new data.

2.2 Two phases of importing the data:

2.2.1 Converting from the portal (xml or text) file to the UPML file

Each portal will have its own drivers to convert from its data file to the UPML file. The drivers for converting from a text file or an xml file were written by me.

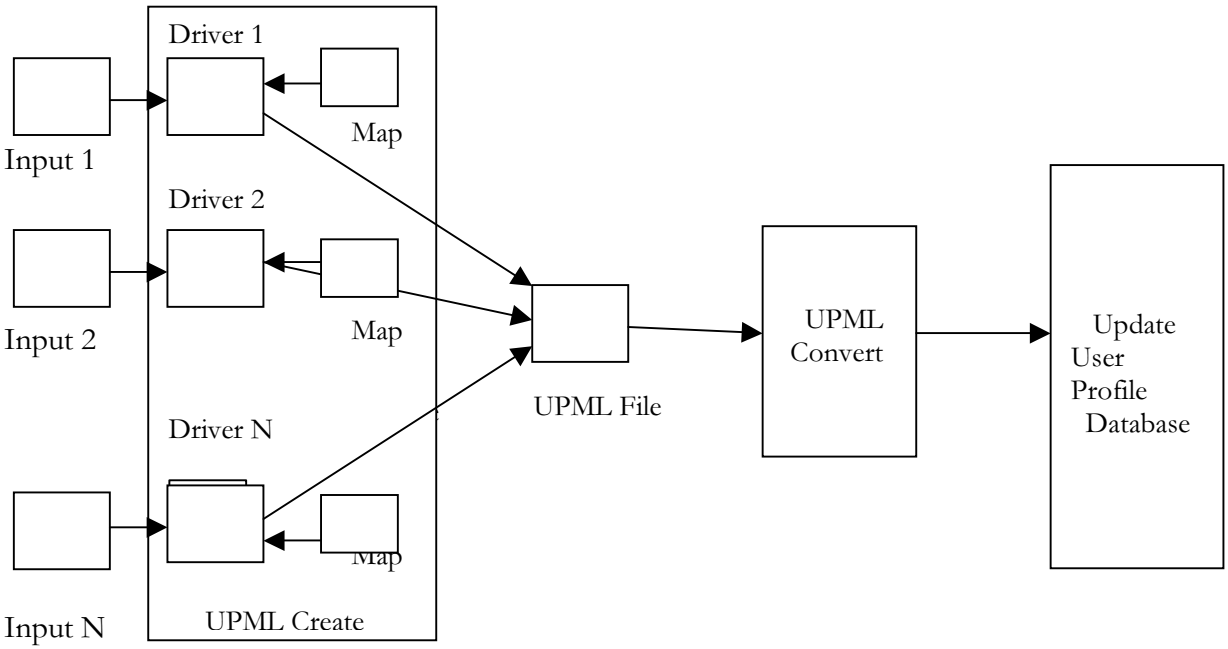
There is also a map file associated with each affiliate. This map file contains three columns. The first and the second columns are the variables names and

their values that the particular affiliate recognizes and the third column contains the respective demogid.
The program UPMLCreate takes the input file from the affiliate's database, the driver for that type of the file and the map file for that affiliate and generates a upml file as output.

2.2.2 Importing the data from a UPML file to the Inktomi database

Then a general program UPMLConvert takes the UPML file as input, parses it and updates the database depending upon the pair <demogid, value> for the particular user.

The block diagram of the whole process is shown below:

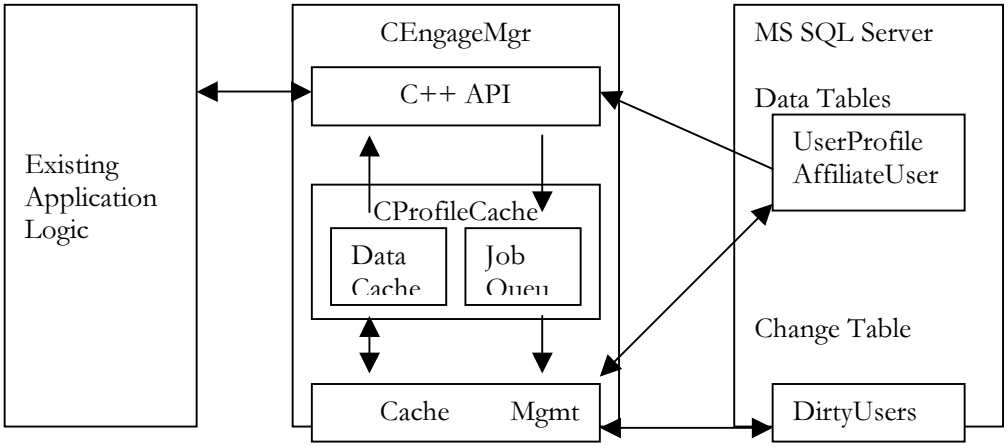


This was an independent project given to me and I completed it with satisfaction.

1. Objective

The profile caching improves the scalability of the user profile system of the TargetServer* in the Inktomi's Selling System by dynamically caching user profile data for currently most active or logged-on users into memory of TargetServer and keeping cached data in synchronization with back-end SQL server database changes. The goal was to scale the TargetServer to be able to efficiently handle 50 million users for targeting and user personalization.

1. Dynamic Data Caching Architecture



TargetServer **Profile DBMS**

* - TargetServer is used to target deals to a particular user depending upon its interests and other demographic values. It is a main part of the Inktomi's selling system.

2. Profile Caching Logics

Without profile caching, whenever an Internet user comes to request data from TargetServer, his/her profile data need be read directly from PROFILE database to target a set of deals for this user, and then a new job need be created and put into the corresponding job queue. Obviously, targeting deals and personalizing user by this way, both MS SQL Server and TargetServer are not scalable if there are millions of

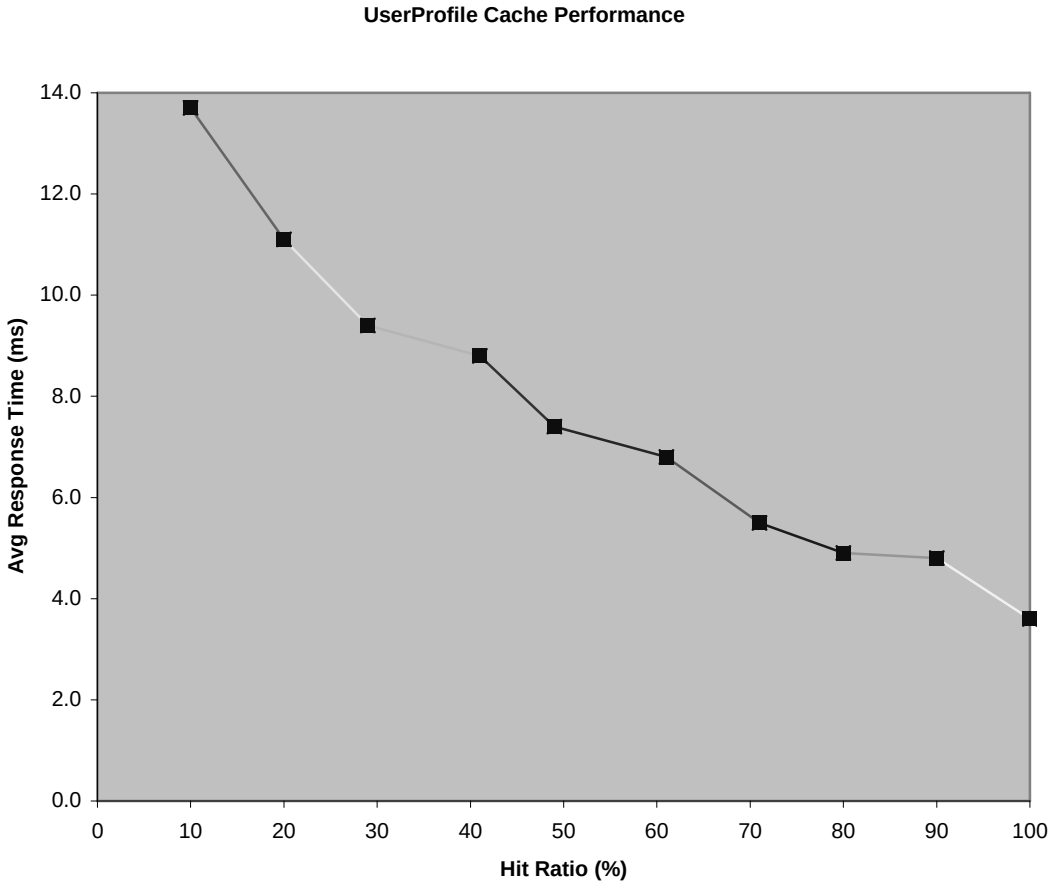
Internet users who access Inktomi's I-merchandising system. So in-memory profile cache for TargetServer has to be implemented.

With profile caching, whenever user profile data are required, they can be read directly from profile cache if they exists in profile cache; otherwise, they are read from profile database and saved into profile cache for future use at the same time. In addition, when an Internet user comes to request data from TargetServer, a new job is created and then put into job queue so that CEngageMgr daemon thread can save user profile data into profile database asynchronously.

The PROFILE database has a table DIRTYUSERS which stores the profile UserIDs or AffiliateUserIDs whose profile data were changed recently, e.g. within 40 minutes. Whenever CEngageMgr daemon server thread saves user profile data in profile database, it will add a new entry or update an existing entry to the table DIRTYUSERS to reflect the changes made in the table UserProfile and AffiliateUser. Then, periodically CEngageMgr daemon server threads of other TargetServers running on different machines will reload those dirty user profile data into their profile caches based on the records in the table DIRTYUSERS and last database access time to reload dirty user data asynchronously and perform timestamp-based replacement of Least-Recently-Used entries in the profile cache if the profile cache is full.

From time to time, by generating new jobs and putting them into the job queues of CProfileCache, CEngageMgr creates new profile users or updates existing profile users, or even read profile user data from profile database if the coming profile user is missed in current profile cache of CProfileCache. In addition, if one profile user is read through profile ReadCache, its timestamp will be updated to current time. By this way, a latest version of LRU entries in profile WriteTime after switching profile ReadCache and WriteCache can be kept.

My contribution:
My job was to test and analyze this system and to find out if the cache actually works. If, a cache is to be tested, then the best graph will be to plot the average response time of a cache access against the hit ratio, keeping the cache size constant at all times. I wrote a program that could take hit ratio as input and return the average response time for a fixed number (1000 in my case) of cache accesses. To fix the hit ratio, I loaded the cache with a known data and then tried to access it for a data within a certain proportion so that I get a fixed hit ratio. The graph for a cache size of 5K is shown below:



So, as we can see, the response time actually decreases as we increase the hit ratio, and therefore, it proves the working of the cache.

BENEFITS FROM THE INTERNSHIP

1. I learnt how to read and understand huge pieces of code and how to design a huge system and make sure it fits in with the existing solution.
2. I worked with Visual versions of C++ and Java, which was a new experience for me.
3. I learnt the importance of keeping backups of code even if a small change is made. For the purpose, we used Visual SourceSafe developed by Microsoft.
4. I learnt new concepts of working with windows registry and concluded that programming in Java is much easier and involves less programming time and skills, though at the same time is computationally expensive that other languages like C++.
5. Had a great experience of working with real fast machines with huge RAMs. The TargetServer had 4 processors with processing speeds in GHz and a RAM of 1.5 GB!
6. It was a great experience overall.