Scalability and Efficiency in Personalized Web Services

by

Osama M. Al-Haj Hassan

(Under the direction of Lakshmish Ramaswamy and John A. Miller)

Abstract

Web 2.0 has been growing at a rapid pace empowering end-users with a vast set of applications dedicated to improve their experience while using the Web. This improvement comes in the shape of increased personalization that enables the end-users to navigate and search the Web based on their own needs. One of the key icons of Web 2.0 applications is mashups; they are essentially Web services that are often created by end-users. They aggregate and manipulate data from sources around the World Wide Web. Surprisingly, research related to mashups performance received little attention in research community. In this dissertation, we provide architectures, protocols, and schemes to enhance mashups performance and scalability. We provide improvement over mashup execution by defining a protocol and a set of rules that change the ordinary mashup execution paradigm. Further, we design caching protocol to utilize data reusability in mashups which results in more efficient mashup execution. Moreover, we propose a distributed mashup architecture which increases the scalability of mashup platforms. All the former techniques and protocols are backed up with a set of experiments proving their effectiveness in transforming mashup execution to a more efficient and scalable process.

Index words: Web services, Mashup, Personalization, Data reuse, Web 2.0, Feeds, Caching, Indexing, Distributed, Performance, Scalability

Scalability and Efficiency in Personalized Web Services

by

Osama M. Al-Haj Hassan

B.Sc., Princess Sumaya University for Technology, 2002

MS., NewYork Institute of Technology, 2004

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2010

Scalability and Efficiency in Personalized Web Services

by

Osama M. Al-Haj Hassan

Approved:

Major Professors: Lakshmish Ramaswamy
John A. Miller

Committee: Thiab Taha
Maria Hybinette

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
July 2010

Dedication

Dedicated to my parents who are my source of inspiration, and to my brothers and sisters who gave me support.

Acknowledgments

First, I thank god for his continuous blessings and for giving me strength to go through and finish my PhD path. With a great pleasure and appreciation, I would like to thank my major advisor Dr. Lakshmish Ramaswamy for his constant support through the past five years; he knew how to give me the best guidance through my PhD by providing valuable directions, helpful thoughts, and continuous encouragement. His thoughtful directions enlightened me with great research ideas and his supervision style that emphasized on freedom of research unleashed my productivity to outstanding limits. I cannot thank him enough for making me a better researcher and scholar. Special thanks to my Co-advisor Prof. John Miller who provided me with insightful comments that enhanced the quality of my research. My committee members, Prof. John Miller, Prof. Thiab Taha, and Dr. Maria Hybinette, many thanks for their useful comments which were of a great help to me through my research, I want to express my heart full gratitude to all of them. My parents, your sacrifice throughout the years is what made me who I'm now, no words can describe my gratitude to you. My father Prof. Mohammad Al-Haj Hassan, many thanks to him for keeping me believing in myself, I always looked up to him as a role model. In the moments of weakness, his image in my head has always charged me with power and energy to go on. My mother, she has the sweetest warmest heart ever, she showered me with her love since I was born, and I want to express my sincere gratitude to her.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our work is targeted towards the new Web 2.0 applications. In this chapter, we define Web 2.0 and we describe the difference between Web 2.0 and the older version of the Web. In addition, we give a classification of Web 2.0 applications and examples on them.

1.1 Web 2.0

Web 2.0 is the new generation of the World Wide Web, it is mainly associated with applications that involve information sharing, personalization, collaboration, and user-centered design [1]. Web 2.0 is a newer more innovated version than Web 1.0, the main difference between the two versions is the amount of end-user participation. In Web 1.0, end-users have a more passive role in which they mostly read published content. On the other hand, end-users role increased in Web 2.0 by having them generate content (end-user generated content). Web 1.0 is about companies that generate content over the Web and end-users read that content. In Web 2.0, the focus shifted from companies to end-users by having end-users generate more portion of Web content. Web 1.0 was popular of its client-server networking model. Now Web 2.0 is more about peer-to-peer networking model. Web 1.0 relies on HTML as the building block of Web pages while in Web 2.0, XML intervened in the picture to bring more semantically enriched content for the Web.

Web 2.0 applications can be classified into social applications, content management applications, blogging applications, service-oriented applications, podcast applications, social bookmarking applications, video processing applications, audio processing applications, photography applications, e-commerce applications, online workspace applications, and feed processing applications. Examples of social applications are myspace [2] and facebook [3] where people connect and collaborate with each other, share documents, images and personal comments. Content management applications are designed to enable people to share information and knowledge while at the same time enriching this knowledge by enabling them to alter information, an example of these applications is Wikipedia [4]. Blogging applications give space for people to express themselves more and to write about their personal opinions and their daily life activities, an example of these applications is Blogger [5]. Podcast applications (podcatchers) are those applications that allow you to check a feed of audio/video files list (podcast) belonging to a specific series, then the files are downloaded by the podcatcher. This process can be automated such that the podcatcher checks the feed for new updates, and download any new files; an example of podcatchers is Juice [6]. Social bookmarking applications allow users to save their bookmarks online for future access, they also allow users to share bookmarks with each other, an example of social bookmarking applications is Delicious [7]. Audio, video, and photography applications provide a way for users to share images, audio and video. In addition, users can tag these files, recommend it, or make discussions about them. Examples of those applications are Pandora [8], Youtube [9], and Flickr [10]. Online workspace applications allow users to arrange their work online, such as online calendars, saving and editing documents online, and online project management, example of this kind of applications is ZOHO [11]. E-commerce applications mash several information from different websites to provide users with products from different sources; this helps users to pick the best deal in one place. An example of this application is KAYAK [12].

Feed processing applications are those applications targeted towards fetching data from several distributed data sources over the Web, one instance of this category is a group of

Table 1.1: A classification and examples of Web 2.0 applications

Application Group	Examples
Social	MySpace, Facebook, LinkedIn, Meetup
Content Management	Wikipedia, Wetpaint, Digg, Wikia
Blogging	Blogger, Superspace, Blogspot, quackit
Podcast	Juice, Podnova, Gigadial, PodcastPickle
Social Bookmarking	Delicious, Socialmarker, Diigo, Reddit
Feed Processing	Google Reader, kedoya, Yahoo Pipes, Intel MashMaker
Online Workspace	ZOHO, Mint, Rememberthemilk, Basecamp
Photography	Flickr, Fotoflexer, Picasa, Photobucket
Audio	Pandora, Ilike, Live365, emusic
Video	Youtube, Joost, Ustream, veodia, Vimeo
e-commerce	KAYAK, Woot, Etsy, veodia, Vimeo

applications called feed readers. A feed reader is a Web or desktop application in which end-users subscribe to feeds, such that the reader periodically checks the feeds for updates; updates are reflected directly on the reader application interface. In other words, feed readers provide a way where end-users are kept updated with feeds information which is all collected in one application. An example of Web readers is Google reader [13]. Other applications go beyond fetching data by aggregating, manipulating, and filtering this data to deliver the end-result to end-users. These applications are called mashups, examples of these applications are Yahoo pipes [14] and Intel MashMaker [15]. Table 1.1 shows a classification and examples of Web 2.0 applications. This classification is based on [16] and [17].

1.2 Feed Processing Applications

One key category of Web 2.0 is feed processing applications. Feeds are basically sources of data distributed over the Web and they are represented in two formats, RSS and Atom. RSS

```
<rss version="2.0" xmlns:media="http://search.yahoo.com/mrss/
"xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:atom="http://
www.w3.org/2005/Atom">
<channel>
   <ttl>10</ttl>
   <language>en-us</language>
   <title>Yahoo! Sports - Tennis News</title>
   <link>http://sports.yahoo.com/ten</link>
   <description>
         Latest news and information about the Tennis.
   </description>
   <item>
    <title>Foot could keep Clijsters out 6 weeks (AP)</title>
    <link> http://somelink1 </link>
    <description>
          Kim Clijsters is out for up to six weeks after tearing a
          muscle in her left foot during a Fed Cup singles match
          Saturday
    </description>
    <pubDate>Sun, 25 Apr 2010 10:51:25 PDT</pubDate>
   </item>
   <item>
    <title>Qualifier ousts Ferrero at Rome Masters (AP)</title>
    <link>http://somelink2</link>
    <description>
          Colombian qualifier Santiago Giraldo unleashed an
          arsenal of winners to upset former champion Juan
          Carlos Ferrero 6-0, 6-3 Monday in the opening round
          of the Rome Masters.
    </description>
    <pubDate>Mon, 26 Apr 2010 07:14:30 PDT</pubDate>
   </item>
   <lastBuildDate>Mon, 26 Apr 2010 07:14:30 PDT</lastBuildDate>
 </channel>
</rss>
```

Figure 1.1: Example of Yahoo Tennis RSS feed

and Atom feeds are designed as XML-Like files that contain tags describing a set of items; each item represents a piece of information. Typically, each item has a title, description, author, and publication date. Although feeds usually refer to RSS and Atom formats, feed processing applications sometimes can handle other sources such as CSV, XML, JSON. An example of a feed is listed in Figure 1.1 which represents Yahoo Tennis feed that lists the latest updates in Tennis news.

Mainly, feeds are consumed by two types of applications, namely, Feed Readers and Mashups. Feed Readers are Web or desktop applications dedicated to reading feeds, collecting them in one user interface, and reflecting feed updates on the user interface. The way they work is by end-users to subscribe for feeds, after that, the feed reader application becomes responsible of pulling these feeds from their hosts periodically, and displaying updates to end-user. The

advantage of such an application is first its ability to combine information that interests end-user in one place which relieves the end-user from visiting several different Websites. Second, it shows the end-user which feeds are updated and that spares the end-user from the burden of checking feeds that have not changed. Example of feed readers is Google reader [13].
Although feed readers are useful applications, they fall short when it comes to manipulating information. Feed readers only provide end-users with a way to read feeds. An application that overcomes this disadvantage is Mashups. Mashups are Web services created by end-users, they enable end-users to fetch data from several data sources distributed over the Web, aggregate, manipulate, and filter this data. Examples of information manipulation in mashups are aggregation, filtering, sorting, and truncation. An example of mashups is a mashup that fetches information from Yahoo sports feed, then it filters the feed for Tennis related items, then the result is sorted. Mashups have several advantages. First, they have the ability to combine data from several sources in one place. Second, this data can be read and also manipulated. Third, it offers high degree of personalization by enabling end-users to refine data based on their own personalized needs. Examples of mashup platforms over the Web are Yahoo pipes [14] and Intel MashMaker [15].

1.3 Contribution of the dissertation

Since the advent of Web 2.0, there has been increasing importance for Web 2.0 related applications. Mashups are one of the key categories of Web 2.0 applications. Surprisingly, performance and scalability of mashups received little attention from research community. This dissertation investigates mashups performance and scalability issues by exploring challenges facing mashup execution in mashup platforms. Based on these challenges, we propose architectures and techniques to ensure efficient and scalable mashup execution. The contributions of this dissertation are pointed out by the following three innovations:

- We present a model for representing mashups and analyzing their performance; this model defines mashup platforms, mashups, and their components. Modeling for component's inputs, outputs, and representation is introduced.

- We design operator merging technique and operator reordering rules. Repeated execution of identical operators leads to deficiency in executing mashups. Detecting identical operators and merging them are necessary so that identical operators are executed once. Mashups are not optimized at design time because they are designed by end-users who are not aware of mashup execution efficiency. Consequently, we provide a set of operator reordering rules that arrange mashup operators in the most optimized order. Both, operator merging and operator reordering lead to more efficient mashup execution.

- We provide mashup caching framework that helps in caching results of execution of parts of mashups. Our caching technique takes into consideration common operators across mashups so that cached data is chosen carefully to increase the value of our cache and to make mashup execution more efficient.

- We design a distributed architecture for executing mashups. Ordinary mashup platforms are based on central server architecture which degrades their scalability. By providing our distributed architecture, we increase the scalability of mashup platforms. Our architecture is based on an overlay of nodes where a mashup is executed on several execution nodes. Our system handles failure resiliency issues by replicating nodes and parts of mashups.

1.4 Organization of the dissertation

In this dissertation, we start by the introduction (Chapter 1) where we define and describe Web 2.0. At the same time, we discuss several important key categories of Web 2.0 applications such as feed readers and mashups. In addition, we describe our contributions in this work. In Chapter 2, we informally define mashups and mashup platforms. In addition, we

discuss related work in several areas that include a discussion of existing mashup platforms, Web performance techniques, personalization, Web content caching, Web services caching, and distributed data processing platforms. Further, we describe motivations and challenges that mashup platforms are faced with. In Chapter 3, we define a mashup model and notations that we rely on in the rest of this dissertation. That includes the entities interacting together in a mashup platform. Also, we show how we represent mashups in our system. In addition, we propose a novel indexing scheme that aids the access of mashup parts. The representation scheme and indexing scheme described in this chapter are going to be used throughout this work. We follow that by Chapter 4 in which we propose new techniques for improving the efficiency and scalability of mashup execution. Those techniques include detection of identical components across mashups which helps to avoid unnecessary repetitive execution. It also includes an operator reordering scheme that helps in redesigning mashups to reflect a more optimized mashup workflow. Chapter 5 continues the discussion on improving the efficiency of mashup platform; this time it is done by proposing a caching framework in which intermediate mashup execution results are stored locally; this helps in avoiding executing the same mashup parts in future requests. We design an efficient dynamic greedy algorithm for selecting data to be cached. In Chapter 6, we propose a cooperative distributed mashup platform where mashup execution takes part on several distributed nodes. We also propose an algorithm for placing mashup parts on network nodes such that the cost of executing mashups is minimized. In Chapter 7, we back up the previous architectures and techniques with an extensive experimental evaluation that finds out the benefits and costs of our techniques. We finish with conclusions in Chapter 8.

Chapter 2

Background and Motivation

In this chapter, we define mashups and mashup platforms. In addition, we give examples of mashups and how they are executed. Further, we discuss related work in the area of mashups, personalization, and Web 2.0 performance. Furthermore, we describe the challenges and motivations that led our research towards working in the area of mashups.

2.1 Background

In this section, we informally define mashups and mashup platforms. Conceptually, a mashup is a Web service created by an end-user to reflect his own personalized needs. The purpose of a mashup is to fetch data from one or more sources over the Web, process this data based on end-user needs, and deliver the results to the end-user. Processing data can be in the form of filtering, truncating, merging, and sorting. All fetching, data processing, and sending output operations are performed via their corresponding operators. For example, filtering is performed using a filter operator that specifies the way data should be filtered.

Mashup platforms are those systems that are responsible of hosting mashups. They have multiple tasks. First, they provide a friendly interface for end-users to design their own mashups. Second, they receive and host mashups submitted by end-users. Third, they are responsible of executing mashups and delivering the end-result to end-users. Fourth, they maintain mashups for future use by end-users. Examples of popular mashup platforms are Yahoo pipes [14] and Intel MashMaker [15].

An example of a mashup is one that delivers to the end-user sports information coming from Yahoo Tennis sports and ESPN sports such that information is related to the Tennis

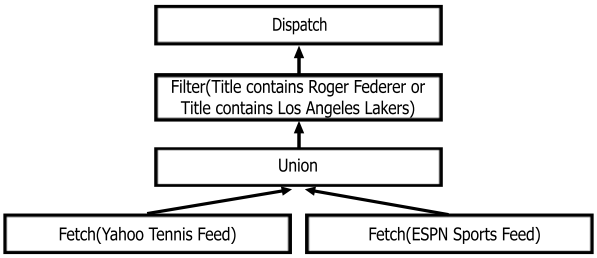

Figure 2.1: Basic representation of a mashup and its components

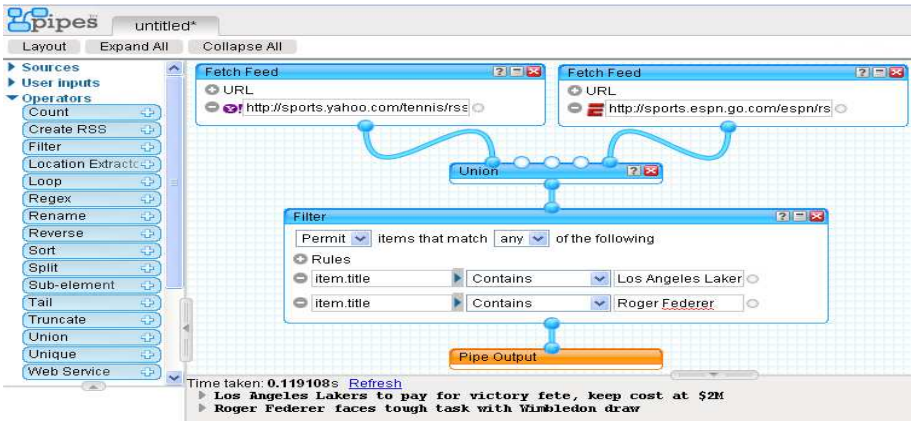

Figure 2.2: Basic representation of a mashup and its components

player Roger Federer and Basketball team Los Angeles Lakers. As shown in Figure 2.1, this mashup uses two fetch operators; the first one fetches data from a Yahoo Tennis feed and the second one fetches data from ESPN sports feed. After that, data is combined using a union operator. Then, data is filtered based on Title contains Roger Federer or Title contains Los Angeles Lakers. Finally, the end-result is dispatched to the end-user. A design of this mashup in Yahoo pipes mashup platform is shown in Figure 2.2.

2.2 Related Work

Research in the area of mashups is still in its nascent stages. In this section, we discuss the current related work in the area of mashups. First, we start by listing work related to mashup platforms. Second, since our work targets the improvement of a Web 2.0 application, we review existing approaches for improving performance of Web applications. Third, because mashups are all about personalization, we discuss related work in the personalization domain. Fourth, one core part of our work is a caching framework for mashups, therefore, we study the related work in the caching domain. Fifth, since mashups are conceptually Web services, we take a closer look at caching for Web services. Sixth, we discuss current distributed data processing architectures and how they are compared to the distributed mashup architecture we propose in this work.

2.2.1 Existing Mashup Platforms

The number of mashup platforms and research conducted in the area of mashups is increasing due to the burst of Web 2.0. Yahoo pipes [14] is the most popular mashup platform on the Web. It handles different types of data sources including JSON, XML, RSS, Atom. It also includes different types of operators that facilitate data manipulation. Their platform also enables end-user to collaborate by reusing each other mashups. MARIO [18] is a recent mashup editing tool in which mashups are built from tags and executed using a planning algorithm. DAMIA [19] is a data integration service for situational applications in the enterprise domain. Kulathuramaiyer [20] describes a mashup for digital journals which enables its users to explore digital libraries using semantic-rich meta-data. Subspace [21] adopts the sandboxing principle to isolate applications into trust layers. Marmite [22] is a mashup tool implemented as a Firefox plug-in using Java script and XUL, it enables end-users to aggregate and filter data from several Web contents and services. It also has the capability of directing mashups output to several sources such as websites, text files, or even compliable source code that can be customized. MashMaker [23] is a mashup Web tool that enables

end-users to manipulate data from other websites and define their visualized queries over it, it also provides sharing of mashups as widgets among end-users. Liu et al. [24] provide a mashup architecture that is based on extension of the Service Oriented Architecture (SOA). Similar to (provider, broker, and consumer) SOA roles, their system presents roles of (mashup component builder, mashup server, and mashup consumer), such that these roles interact together to facilitate data and services composition to end-users through a Web based interface. Karma [25] is another mashup platform that enables end-users to build mashups by example. Karma authors argue that using widgets as building blocks of mashups is not convenient for end-users because as a mashup platform grows, the number of widgets grows as well and that will confuse the end-user on which widget he needs to perform his task. Based on that, the authors let end-users extract data they want from websites. The data is stored behind the scenes as DOM trees and it is visualized to the end-user as tables, then the end-user can visually work on the data with the help of their system. Their system allows end-users to fix and integrate data and define rules between attributes of data, these attributes are stored in the system as XPath rules and the system intelligently applies the rules on data containing the previous attributes. Smash [26] is a security model that can be integrated in Web browsers in order to make mashup applications secure. Authors state that current browsers security models are not appropriate for mashups because mashups require interaction between different data sources while regular browsers usually disallow such interactions between links coming from different sources. Smash defines a security model so that scripts coming from different data sources are not allowed to change each other's data. At the same time, the model does not allow such scripts to spy on end-user data banks. The model applies component isolation in mashups while at the same time guaranteeing communication between components using authenticated communication channels. The model also defines security rules on which types of interactions are allowed between components. In [27], authors look at the operational challenges facing mashups. Since mashups combine functionality from different organizations, difficulties regarding communication and role of each organizational

part in the mashup may arise. In [28], design principles of enterprise mashups are discussed, the role of end-users in enterprise mashups is described, and an explanation of why current service oriented protocols such as SOAP fail in the case of enterprise mashups.

2.2.2 Web Performance

Web performance is an area that has been extensively studied in research. Many factors affect the performance of Web applications such as delay, bandwidth consumption, reliability, and load balancing. Delay is one of the most important factors because end-users who use Web applications demand applications with minimum delay. Bandwidth consumption is an important resource that has to be conserved. Bandwidth consumption and delay are tied together in a sense that minimizing one of them minimizes the other. Delay results from communication time and computation time but communication time is most likely the dominant factor in delay. Caching is used to minimize communication delays and bandwidth consumption by storing data locally and using it for further end-users request. Caching schemes [29] [30] [31] [32] can be in the form of edge caching schemes, cache routing schemes, multicast-based schemes, and directory based schemes. More details about caching techniques is presented in sections 2.2.4 and 2.2.5. Detecting common components is another way to minimize delays and bandwidth consumption. Many Web applications contain identical components that are repeatedly executed, if such components are not detected, we end up with unnecessary execution of identical components. This is extremely important in the cases where bulks of operations are executed at the same time. Detecting common components has been investigated in many systems such as [33] where improvement on flooding technique is proposed to reduce the number of duplicate messages in peer to peer networks. It is also used in mobile broadcast systems [34] to eliminate redundant messages. Common component detection has been used in tree structures. Tree isomorphism for ordered and unordered rooted trees is discussed in [35] [36]. Detecting common content in XML document trees is discussed in [37]. Efficient routing is also used to minimize communication

delays. efficient routing in peer-to-peer networks is studied in [38] where authors propose routing algorithms based on number of hops which lead to small lookup paths. Ratnasamy et al. [39] discuss open problems in DHTs which are mainly used in peer-to-peer systems. Multicast routing algorithms are surveyed in [40]. Aggregating messages to be sent through the network is usually a tradeoff between increasing delay and minimizing bandwidth. As in [41] [42], this happens because usually if the header part of the message is much larger than the body part, aggregation makes sense because communication saving is higher than delay overhead. In contrast to the previous case, if the body of the message is much larger than the header part, then there is not very much saving in data size by aggregation. In this case, saved communication is less than delay overhead which makes disaggregation a better choice. One point to mention is that in real time systems, aggregation might not be adequate because of the stringent time constraints. Data compression is necessary to minimize delays and bandwidth consumption in cases of heavy transmission of data such as in the case of video transfer. Compression techniques related to image compression, audio compression, video compression, wavelet compression are presented in [43]. Data compression in text retrieval systems is discussed in [44] where authors provide a transparent layer that uses Huffman code to apply data compression. Query optimization is also popular for more efficient Web applications. Ouzzani et al. [45] provide a survey of query optimization on the Web in addition of research issues in the area of query optimization. Reliability is one of the important assets of Web applications. The report in [46] investigates and classifies sources of failure in Web application. Techniques to ensure reliability are proposed in literature such as [47] [48]. Reliability can also be provided by increasing the availability of the system using replication. When multiple nodes in a network are replicated, one can serve instead of the other in cases of failures. Also, replication of resources such as files increases their availability on the Web. Replication techniques are proposed in [33] [49] [50]. Load balancing is a criterion that helps to distribute load on processing nodes in a network, it produces better

utilization of nodes in addition to minimizing their probability to fail. Many load balancing techniques are discussed in literature such as [51] [52].

2.2.3 Personalization

Since Web 2.0 is all about end-users, Web 2.0 main feature is the focus of personalizing end-user experience when using the Web. Personalization is the process of automatic dynamic customization of Web sites, end-user queries, and end-user related applications based on their own needs. Personalization challenges are presented in [53] in the context of recommendation systems and Web searching. Recommendation systems are systems that give users recommendation of products or items of interest to them. It is popular in e-commerce where personalization comes into picture by recommending items to users based on their personal interest. Personalization in such a system can be achieved by content-based filtering which basically captures aspect of items, and it can be achieved by collaborative filtering which works on the assumption that if user x is interested in a set of items, and user y is interested in similar set of items, then items by user y can be recommended to user x. In the context of Web searching, personalization can be achieved by defining end-user profiles, tracing end-user search activity, and by using semantic Web means such as meta data and taxonomies. Mobasher et al. [54] specify the elements of Web personalization, namely, modeling of Web objects (such as pages) and subjects (end-users), categorization of these objects and subjects, matching between those objects and subjects, and deciding what actions are needed for personalization. According to Srivastava et al. [55], types of Web data that can be used for personalization are content, usage, structure, and user profiles. A classification of Web personalization techniques is given in [56] where techniques are classified into the following groups; user profiling, log analysis, Web usage mining, content management, Web site publishing, and information acquisition and searching. Datta et al. [57] propose a replacement for static profiles, they propose the use of dynamic profiles to enhance the personalization of Web sites. A dynamic profile is information collected for the purpose of predicting the future

actions of end-users. Authors also provide an architecture that is specifically designed for e-commerce applications. Using association rules to detect user patterns in his Web activity is discussed in [58]. Authors provide an efficient data structure for saving the items visited by users, their data structure is in the form of a sliding window which contains the last n visited items by users. Authors design an algorithm to use this data structure to produce real-time recommendations. The same data structure is used to discover association rules between frequent items. Yoda [59] is a hybrid system that combines content analysis, clustering, and collaborative filtering features. It also uses recommendation lists by experts such as user experts. It also uses navigation patterns by users to give better recommendations. Mobasher et al. [60] define URL clusters which are clusters of URL references computed based on their appearance patterns in user transactions. In their system, Web server records HTTP requests performed by users during their sessions, the user sessions and URL clusters are considered together to compute a set of suggested URLs that users might be interested in. Good et al. [61] proposes a hybrid system that combines information retrieval(IR) with collaborative filtering(CB), they achieve that by considering user opinions and ratings in addition to Web agents (Filterbots) who create profiles for users and generate predictions based on those profiles. Liu et al. [62] propose an approach for using user profiles and categories to modify user queries in a way that makes user queries more meaningful and personalized. The user profile is built based on tracking user navigation history. Sometimes, users might include new terms in their queries, in such a case, user profiles do not help. Accordingly, authors use category profiles which are public profiles containing information about categories and keywords related to all users. If a user is using a new term, then it can be found in the category profile and the user query is updated consequently so that the meaning(context) of the new keyword becomes clear. Henze et al. [63] design a personal Web reader framework which provides personalized content for individual users. In their work, authors use different ontologies to bring meta data enrichment to the Web reader. The types of ontologies they use are domain ontology, user model ontology for user characteristics and preferences, observation

ontology for describing user observations, and adaptation ontology for describing adaptation processes. In addition to ontologies, they use personalization services for adaptation from hypermedia field. The ontologies combined with personalization services help in adaptation of user Web content and it helps in providing users with recommendations relevant to their Web content. Cingil et al. [64] propose an approach for using W3C standards to provide personalization for Web users, they use a client side agent that captures user click stream and store it in a log coded in XML. The user log size is huge; therefore, authors use XML-QL queries combined with navigation statistics to build a user profile coded in RDF. In order to maintain user profile privacy, Web servers can only access user profiles through a Platform for Privacy Preferences (P3P). Web servers access user profiles to provide Web users with personalized content based on similarities of interest among users.

2.2.4 Web Content Caching

Web content caching in general has received considerable research attention [29] [30] [31] [32]. A survey of Web caching on the Internet can be found in [30]. Issues such as caching granularity, caching architectures, consistency maintenance and data placement and replacement strategies have been extensively investigated. Iyengar et al. [29] provide a cache dedicated for caching dynamic Web pages. Caching dynamic data can be challenging especially when the dynamic data changes very frequently. Their approach focuses on minimizing CPU time for servers generating dynamic data rather than minimizing network traffic. They consider the fact that dynamic Web pages contain data (such as images) which do not change quite often. Ramaswamy et al. [31] propose a scheme for fragment caching in dynamic Web pages where they design an architecture and an algorithm for detecting different parts of Web pages (fragments) based on how often a fragment is shared across different pages and also based on the lifetime characteristics of fragments. Yin et al. [32] discuss cache consistency and invalidation issues for static and dynamic Web content. In their experiments, they compare server driven consistency maintenance to pull based approaches. They state that server driven consistency

protocols are the best in the situation where dynamic data is cached. In this case, server side invalidation messages is an efficient approach. Various caching techniques are proposed to optimize several parameters in Web caching. Cache routing schemes are discussed in [65] [66] where hashing is used to map files to cooperative cache groups. Multicast-based schemes [67] are proposed where communication and coordination between caches are performed using IP level multicasting. Directory-based schemes [68] [69] are described where subset of contents of cooperative caches is stored in every cache. Ninan et al. [70] propose a lease based scheme called cooperative leasing in which a lease based technique is used to maintain documents consistency. Gao et al. [71] propose a scheme for edge caching where different levels of object consistency have been taken into account in order to minimize consistency maintenance costs. A dynamic scheme for disseminating data is presented in [72] where each data object has its own dissemination tree which is built based on coherency requirements of caches. Hierarchal caching techniques are discussed in literature, where multiple levels of caches exist in the network such that if user request is not satisfied by the first level cache, then the request is forwarded to the next cache level. An example of hierarchal caching technique is in [67]. Distributed caching is the type of caching where data is cached in multiple distributed caches such that these caches contain meta data describing the contents of each other cached data. This way, these caches can cooperate with each other to serve user requests. An example of distributed caching is the work performed in [69]. A comprehensive analysis between hierarchal and distributed caching can be found in [73] where several parameters are taken into consideration during analysis such as cache latency, bandwidth usage, and cache load. Hybrid caching is a combination between hierarchal and distributed caching. In such an architecture, caches can cooperate with other caches in the same level (distributed) or in higher or lower levels (hierarchal). One example of a hybrid caching architecture is [74]. Cache consistency is one of the most important issues for caching. According to [75], cache consistency schemes are classified into pull based schemes and push based schemes. In pull based schemes, client needs to pull data from its origin. Examples of these schemes are using TTL, lease based,

client periodic polling, and client pulling every time. On the other hand, push based schemes are the schemes in which the server is responsible of providing client with up-to-date data to keep their cache consistent.

2.2.5 Web Services Caching

Web services, in general, have been a hot area of research in which aspects such as description, discovery, composition, and efficiency of Web services are addressed [76] [77] [78]. Web services caching in particular gained interest from research community. WReX [79] is a caching middleware architecture for caching XML Web services responses. Terry et al. [80] discuss caching XML Web services for mobile clients. Caching Web services by utilizing proxy caches is used to minimize communication in mobile ad hoc networks [81]. Dorn et al. [82] propose a scheme to maintain Web services availability in cases of communication link interruptions; their scheme utilizes a cache proxy to increase the availability of Web services. Ramasubramanian et al. [83] explain that WSDL which is used to describe Web services lacks information to support caching, so authors extend WSDL to include information about what operations are cacheable in a way that is transparent to clients and servers. Challenges on caching Web services in PDAs are discussed in [84] where authors provide a caching scheme that helps alleviating problems resulting from loss of connectivity and their scheme is adaptable to bandwidth changes. In their work, they propose dual cache architecture for caching Web services on PDAs. Due to connection problems in wireless connection, this dual cache architecture utilizes two caches, one local cache at the PDA side and one service provider cache at the server side. Coordination between the local and the server cache is needed to ensure the delivery of Web services messages to the client. Devaram et al. [85] propose caching Web services messages sent by clients. Their idea depends on the principle that every time a client is requesting a Web service, his request should be transformed into a SOAP message. If the client requests the same Web service many times, then a considerable time is wasted on generating the SOAP message payload. Therefore, they propose caching

the SOAP message payload in a local cache at the client side so that the same SOAP message is fetched from the cache and sent to the server whenever the client requests that Web service. Takase et al. [86] describe a Web service architecture in which caching is used to cache Web services responses. They state that Web services responses coming from servers require parsing by XML readers such as DOM, the result of parsing can be in a form of DOM tree object. This parsing process takes time, so, in their caching architecture, they cache the post-parsing result (eg. DOM tree object) of Web services responses. This way, Web service responses parsing time is avoided. Mahdavi et al. [87] propose a caching mechanism for Web services in e-business Web applications. Their architecture consists of users requesting a service from a Web service portal; the portal collaborates with Web service providers to satisfy user requests. Authors state that Web service providers know the best about which services responses should be cached. Therefore, Web service providers offer portals a worthiness value that is between zero and one and it indicates how worthy it is to cache a Web service response. The portal keeps a lookup table that contains cached responses associated with time stamps. The portal needs to collaborate with service providers to ensure the freshness of cached Web services responses. Doulkeridis et al. [88] propose a caching protocol for mobile devices in peer-to-peer network. In this work, authors focus on the problem of service discovery in a given user device context. The network space is divided into zones where an administrator is responsible of devices in its zone. Those administrators also cache the list of service provides based on user context. In other words, when a device sends a query searching for a service, the query is associated with the device context, such as the device location, and based on that context, a list of services that satisfies user request are selected. Further, the user query and its context are cached so that they are used for future requests. However, as we remarked earlier, most of the existing Web service caching schemes store results at fixed stages of the Web services, and hence are less effective for the mashup domain.

2.2.6 Distributed Data Processing Platforms

Several platforms for distributed component execution have been investigated in literature. A platform for distributed stream processing is proposed in [89] where providers publish streams and end-users subscribe for those streams. Each stream is processed in several broker nodes in an overlay network; authors propose a scheme for placing stream processing operators. The cost metric we propose in Chapter 6 (CoMaP) is more comprehensive than the metric they use. Also, we handle failure issues in our work while they do not handle it. Another operator deployment environment is proposed in [90] where deployment is based on DHT routing. However, as explained in [89], using DHT for selecting nodes where operators are deployed can lead to bad node selections. This is because DHT focuses on minimizing number of hops between nodes but not necessarily minimizing delays. Borealis [91] is a distributed stream processing system, operator deployment in their system does not consider network overlay changes such as changes in links delay and bandwidth. Medusa [92] is another distributed stream processing environment that deploys operators in a way to achieve load balancing. Analysis for node placement in overlay network is investigated in [93], however, they focus on analysis of placing machines on network infrastructure while CoMaP focuses on mashup operators placement in overlay network. TAPIDO [94] is a programming model that handles security for distributed object processing systems such as Web services and mashups. Applications such as Web services and mashups require service composition and data aggregation which involve authentication, authorization, delegation, and trust issues. All these security issues are handled via TAPIDO model which is based on Java remote object model. Bonfils et al. [95] propose an operator placement model for distributed query processing, they rely on nodes searching their neighborhood to place operators efficiently in the network, their system also uses operator switching which enables operators to switch their location to another node that results in lower cost. Papaemmanouil et al. [96] is a similar work to [95], it depends on local search within node neighborhood to place operators in the network, in addition they use a more generic cost function for defining the cost of hosting an operator on a specific

node, they also use operator replication and partitioning which provides parallel processing that leads to more efficient query execution. Srivastava et al. [97] propose processing queries in intermediate nodes such that filter operators are placed on nodes that minimize total delay.

2.3 Motivations and Challenges

Mashups, while enhancing personalization and end-user participation, also introduce new scalability and performance challenges. Unfortunately, these issues have received little attention from the research community. Although there have been some studies on the performance of traditional orchestrated Web service processes [98], to our best knowledge, no studies have studied the performance characteristics of mashup platforms or proposed techniques for improving the same.

Although mashups are conceptually Web services, several differences exist between them. First, mashups are designed by end-users. This implies that mashups are highly personalized based on end-user needs. On the other hand, Web services are designed by Web developers who create mashups for a certain class of users; this means that Web services are less personalized than mashups. Second, since mashups are designed by end-users, mashup platforms typically host several thousand distinct mashups, whereas the number of distinct Web services in a typical Web services portal is relatively small. The frequency of execution of most individual mashups is expected to be modest (the request rate experienced by the mashup platforms may still be very high due to the large number of mashups they host). Thus, the data generated in a mashup platform is orders of magnitude greater than its Web services counterpart, whereas the opportunity for data reuse is much lower. Third, mashups fetch data from large numbers of diverse data sources distributed across the Internet. These data sources vary widely with respect to the characteristics of their data - while some data from some sources changes very frequently others may remain unchanged for considerable durations. Furthermore, the data sources exhibit significant heterogeneity with respect to their

Table 2.1: Differences between Web services and mashups

Comparison Criteria	Web Service Portals	Mashup Platforms
Created By	Developers	End-users
Number of distinct Web services/mashups	Several hundred	Several thousand
Degree of personalization	Limited	High
Adherence to design guidelines	Strong	Weak
Ease of data reuse	High	Low
Optimized at design time	Typical	Rare
Data sources	Mostly internal	Mostly external

popularity, performance and reliability. This implies that the costs of executing mashups depend upon external conditions upon which the mashup platform has little control. Fourth, mashups are designed by non-technical end-users who are likely not aware of the efficiency and performance implications of their design. Hence, mashups are less structured and it is unrealistic to expect mashups to be optimized from a performance standpoint. On the other hand, Web services are authored by professional developers; therefore, they are optimized for performance, and they usually adhere to certain broad guidelines with respect to their overall structures. One point to mention is that most current mashup platforms are based on centralized architectures; this increases the probability of system failure. Table 2.1 summarizes the differences between Web services and mashups.

The previous differences between mashups and Web services imply that special attention has to be given to mashup platforms in order to improve their performance. In the next chapter, we lay out notations that will be used throughout this dissertation.

Chapter 3

Model and Notations

In this chapter, we present the definitions and common core that are used throughout this dissertation. First, we define a mashup model. Second, we design a scheme for representing mashups. Third, we propose a novel indexing structure that facilitates accessing mashups.

3.1 Mashup Model

In this section, we develop a formal model for mashups. A mashup platform can be thought of as a system that fetches data from sources that are distributed across the Internet, processes the fetched data in ways specified by the end-users, and dispatches the processed data to the end-users who again are distributed over a wide-area network. $MpSet = \{Mp_0, Mp_1, \ldots, Mp_{N-1}\}$ represents the mashups existing in the mashup platform at a given point in time.

The mashup platform includes a set of basic processing operator classes such as *filter*, *sort*, *join*, *truncate*, *count*, *location-extraction*, *reverse*, *subelement*, *tail*, and *unique*. For ease of modeling, we introduce two special operator classes. The *fetch* operator class corresponds to the function of retrieving data required for a mashup from an external or an internal source, and a *dispatch* operator class represents the function of dispatching the mashup results to the end user. $OpSet = \{op_0, op_1, \ldots, op_{M-1}\}$ denotes the set of operator classes available in the mashup platform. Without loss of generality, operator class op_{M-2} and op_{M-1} correspond to the fetch (represented as fo) and dispatch (do) operator classes, respectively. The rest of the $OpSet$ elements are data processing operator classes. Each operator class may specify certain requirements on the number of inputs that are fed into it and the type and formats of these

inputs. Also, each operator class always produces the same type of output. For example, the sort operator class expects a single table with possibly multiple rows and columns as input, and produces a table with the same number of rows and columns as output.

Mashups comprise of a set of operator instances chosen from the *OpSet* operator classes. Every mashup contains one or more instance of the fetch operator and one instance of dispatch operator. Specifically, a mashup is modeled as a *tree* with each node corresponding to a mashup operator instance. In this tree, the output of an operator node forms (one of the) inputs of its parent node. Furthermore, The dispatch operator instance always forms the root of the tree, and each leaf node corresponds to a fetch operator instances. $\{nd_0^l, nd_1^l, \ldots, nd_{Q-1}^l\}$ represents the nodes in the tree of the mashup Mp_l, where each node corresponds to an operator instance from *OpSet*. We note that while an individual mashup is modeled as a tree, multiple mashups might share data sources, thus forming directed acyclic graphs (DAGs). Although some mashups update the original data sources, this dissertation focuses on those that process data from remote sources rather the ones that update them. For ease of presentation, in the rest of this dissertation, we refer to an operator instance as simply operator.

Each operator in *OpSet* is associated with two functions. The *cost function*, represented as $CF^{op_j}(s_0, s_1, \ldots, s_{q-1})$ for operator op_j represents the cost of performing the operation. The parameters $s_0, s_1, \ldots, s_{q-1}$ represent the sizes of the inputs to the operator op_j. The concept of cost function is generic, and it can be measured in a variety of ways including latency involved in performing the operation and the computational/communication load imposed by the operation. Here, we quantify the cost of an operator through its latency. The output size estimation function, represented as $OSF^{op_j}(s_0, s_1, \ldots, s_{q-1})$ captures size of the output of the operator op_j, where $s_0, s_1, \ldots, s_{q-1}$ are the sizes of the inputs. The *cost value* of a node nd_i^l (denoted as $CV^{nd_i^l}$) in the mashup Mp_l is the value of the cost function of the corresponding operator on the specific inputs indicated in the mashup tree. Similarly,

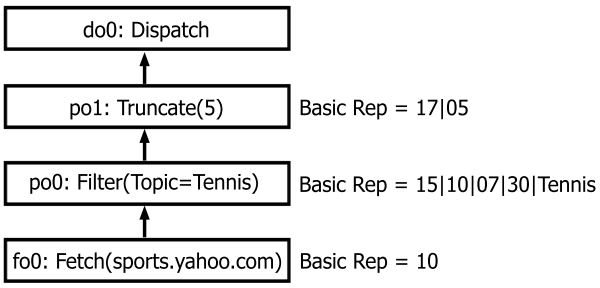

Figure 3.1: Basic representation of a mashup and its components

the output size value $OSV^{nd_i^l}$ of the node nd_i^l is output size function of the corresponding operator evaluated on the inputs specified by the mashup tree.

The total cost of executing the mashup Mp_l is the sum of the cost values of all its operators. Similarly, the output size of mashup Mp_l is OSV of its root node.

3.2 Mashup Representation and Indexing

A mashup takes a form of tree structure, such that mashup execution starts at leaf level by the fetch operators which fetch data from several sources over the Web. Mashup execution ends at the root level by the dispatch operator which sends result to the end-user. Several operators in between the root level and the leaf level exist which process data and refine it based on end-user needs, such as filter and sort operators. Each component in a mashup has two types of representations, 1) basic representation 2) detailed representation. The basic representation defines the operator and it is constructed by concatenating the representation of its attributes such that the symbol | is used as a separator between attributes. Consider the mashup in Figure 3.1, the first operator is a fetch operator that pulls data from sports.yahoo.com; the number 10 in the basic representation is the ID of the data source sports.yahoo.com. The second operator is a filter operator that filters data coming from sports.yahoo.com such that topic equals Tennis. The basic representation of this operator

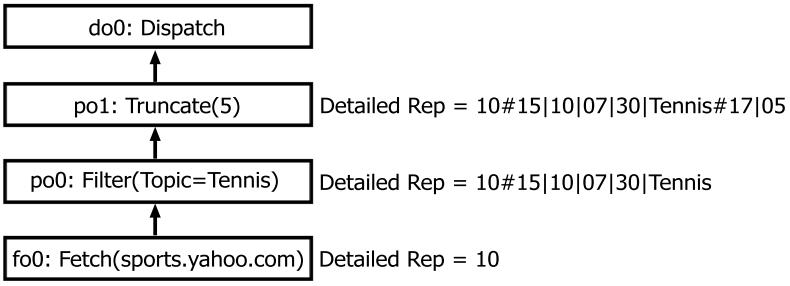

Figure 3.2: Detailed representation of a mashup and its components

is 15|10|07|30|$Tennis$ where 15 is ID of the filter operator, 10 is the ID of the data source sports.yahoo.com, 07 is the ID of the attribute topic, 30 is the ID of the operation equals, and $Tennis$ is the value on which topic values are filtered. The third operator is a truncate operator which has the basic representation 17|05 where 17 is the ID of the truncate operator and 5 is the number of items to truncate from the operator's input.

On the other hand, the detailed representation of an operator is used to reflect the location of the operator in its mashup. This is important because we need to preserve the order of execution of operators. For example, a filter operator basic representation by itself does not imply anything about its location within its mashup and it does not show when it is going to be executed. Therefore, the detailed operator representation is formed by combining the basic representation of the operator with the basic representation of the operators that precede it. Figure 3.2 contains the same mashup in Figure 3.1 and the detailed representation of each operator. Since the truncate operator is executed after executing a fetch and a filter operations, the detailed representation of the truncate operator is $FetchRep\#FilterRep\#TruncateRep$ where $FetchRep$, $FilterRep$, and $TruncateRep$ are substituted by the basic representations of the corresponding operator. The # sign is used as a separator between operators basic representations. Therefore, the detailed representation of the truncate operator is 10#15|10|07|30|$Tennis$#17|05.

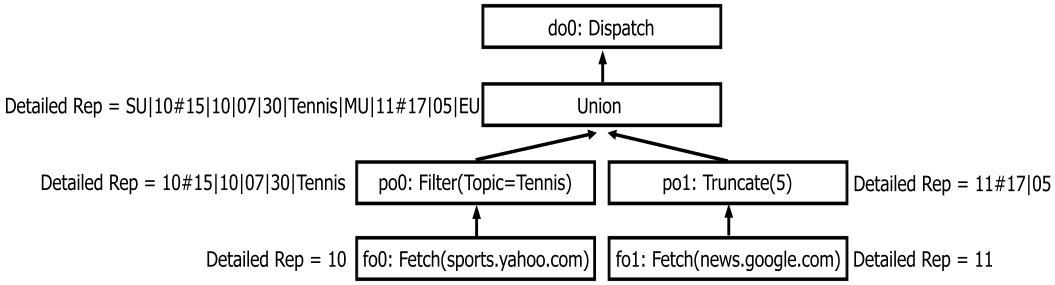

Figure 3.3: Detailed representation of a mashup and its components which include a union operator

Similar basic and detailed representations are used for the operators employed in our system, namely, fetch, union, filter, sort, unique, subelement, tail, truncate, count, and reverse. Unlike other operators, union operators have two inputs representing the two components to be combined and that makes its basic and detailed representations a little bit different than regular operators. Union operator's basic representation is identical to its detailed representation. A union operator starts with special character SU: starts a union block, followed by the first component detailed representation, followed by a special character MU: comes in the middle between the components to be combined together, followed by the second component detailed representation, followed by a special character EU: ends a union block. For example, a union operator that combines two fetch operators fetching data from sports.yahoo.com and news.google.com is represented as follows, $SU|10|MU|11|EU$ where 10 represents fetching data from sports.yahoo.com and 11 represents fetching data from news.google.com. Nested union operators are used to form different variations of mashup structures. Figure 3.3 is a more complex example of a mashup that contains a union operator; the Figure shows detailed representation of mashup operators.

Since mashups can be shared by multiple end-users, we do not include the dispatch operator representation. We do this to keep mashup representation general which facilitates sharing between multiple end-users. A mashup is represented by concatenating the basic representa-

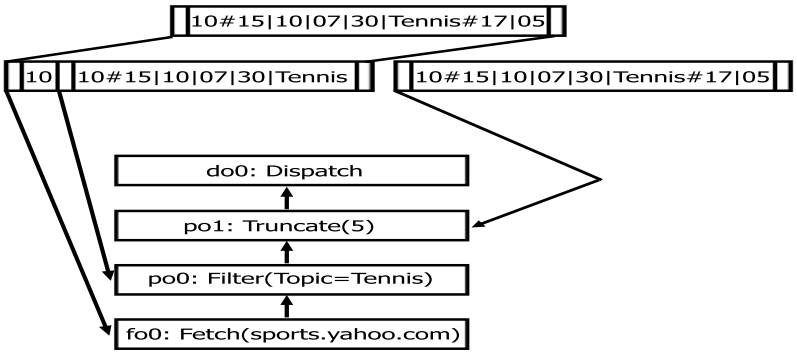

Figure 3.4: A mashup and how it is stored in the components index

tion of each of its components where the symbol # is used as a separator between operators. Since we do not include a representation for dispatch operators, a mashup representation equals to the detailed representation of the operator that is executed right before the dispatch operator. Consequently, the mashup representation of the mashup in Figure 3.2 equals to the truncate operator detailed representation which is 10#15|10|07|30|$Tennis$#17|05. In Figure 3.3, the mashup representation equals to the detailed representation of the union operator.

Since a mashup platform potentially contains large number of distinct mashups, accessing mashup components is inefficient operation. In order to make the process efficient, we use a B+ tree index for mashup components indexing. The index keys are component's detailed representation. The index keys at the leaf level point to mashup components. Figure 3.4 shows a mashup example and its components stored in the B+ tree components index.

After explaining this core information, we continue our discussion with the first piece of improvement we propose on mashup platforms, this piece considers detecting repeated mashup sequences and operator reordering as means of improving mashup platforms efficiency.

Chapter 4

Improving Performance of Mashups Execution

Towards addressing efficiency and scalability issues of mashup platforms, we present the design and evaluation of *AMMORE* - an *A*utomative *M*ashup *M*erging and *O*perator *RE*ordering platform. In designing *AMMORE*, we explicitly consider the fact that mashups may have been developed by multiple end-users with varying-levels of technical expertise. In order to overcome this challenge, *AMMORE* provides the following unique features.

- With the objective of avoiding wasteful repetitive computations, *AMMORE* efficiently detects common components (operator sequences) from different mashups, executes them only once and uses the results in final computations of various mashups. When mashups are executed without common component detection, the mashup platform executes every occurrence of common components which is clearly unnecessary. If common component detection (CCD) is used, some mashups can be merged with other mashups and that minimizes the number of operators a mashup platform has to execute. For example, consider having two *identical* mashups, the two mashups fetch data from sports.yahoo.com then filter data based on topic equals Tennis. Four operators exist in these two mashups. Now, if common component detection (CCD) is used, these two mashups will be merged into one mashup resulting in only two operators. As we can see, the advantages of common component detection (CCD) are two fold. First, the number of components needed to represent mashups in the system is minimized which saves memory space. Second, redundant execution for mashup components is avoided which in turn minimizes delay of mashup execution.

- *AMMORE* converts each mashup into its most efficient form by re-ordering the operators such that the operators that reduce data are executed early-on. Converting mashups into a pre-specified form also improves the effectiveness of common component detection. A mashup execution depends on the order of operators forming that mashup. Consider two mashups, the first mashup fetches data from sports.yahoo.com then performs a filter operation followed by a sort operation. The second mashup is exactly the same as the first one except that the sort operator precedes the filter operator. The two mashups are equivalent because they generate the same output. If order of execution of operators is not considered here, then these two mashups would be treated as two different mashups. Another example of the importance of operators ordering is related to order of siblings in mashup trees. Sibling nodes in mashups can be found in only one case which is the case of having a union operator where siblings are the two inputs of the union operator. Consider two mashups, the first mashup fetches data from sports.yahoo.com and then combines it with data fetched from news.google.com. The second mashup performs the same operation but in reverse order, that is fetching data from news.google.com then combining it with data fetched from sports.yahoo.com. These two mashups can exist because end-users do not follow any specific rules when designing mashups. Although these two mashups look different, they are equivalent. Therefore, the order of siblings for union operators has to be taken into account in order to detect equivalent mashups sequences.

We have developed a real *AMMORE* prototype to the purpose of improving the efficiency and scalability of executing mashups.

4.1 System Architecture

Figure 4.1 shows the components of *AMMORE* which is co-located with mashup platform in the mashup server. The Figure illustrates how our system interacts with the mashup platform. End-users typically design mashups using a mashup editor. Those mashups in addition to

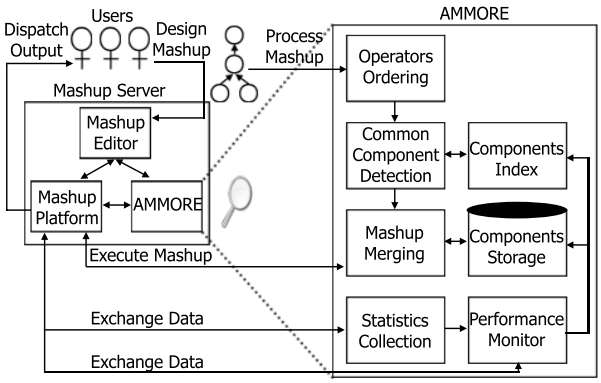

Figure 4.1: System Architecture showing *AMMORE* components and mashup platform

existing mashups are inputs for *AMMORE*. These inputs are taken by *AMMORE* to apply operators reordering on them. After that, the reordered mashups are fed to our common component detector which utilizes the mashup components index described in Section 3.2 to detect common components across mashups. After common components are detected, they are taken as an input for our mashup merger where common components are merged. Following mashup merging, the mashup platform executes merged mashups by coordinating with *AMMORE*. The mashup platform and *AMMORE* exchange housekeeping information to monitor and maintain system performance.

4.2 Common Component Detection

As explained in Section 4.1, mashup operators are represented as strings. As a result, mashup trees are represented as strings which also means that mashup subtrees are represented as strings as well. Detecting common components is equivalent to detecting subtrees in mashups. We are looking to find the longest common operator sequences, because this causes mashup tree representation to be more compact and saves unnecessary computation when executing mashups. Since operator sequences are represented as strings, our problem is to find the longest common substring in mashups string representations.

4.2.1 Mashup Common Component Definitions

Based on the previous challenges listed in Chapter 2, we describe the following definitions to make common component detection problem for mashups more concrete. In order to detect common components across mashups, we need to define the term *COMMON* for two mashups.

Mashup platform has a set of mashups $\{Mp_0, Mp_1, \ldots, Mp_{N-1}\}$. Each mashup Mp_l tree consists of set of nodes $\{nd_0^l, nd_1^l, \ldots, nd_{Q-1}^l\}$. Each node corresponds to a specific operator. Two nodes nd_x^0 and nd_y^1 are *COMMON* across mashups Mp_0 and Mp_1 if and only if:

- Condition1: Mashup Mp_0 contains node nd_x^0 and mashup Mp_1 contains node nd_y^1.
- Condition2: Node nd_x^0 has attribute list $AtList - nd_x^0 = \{At_0^{nd_x^0}, At_1^{nd_x^0}, \ldots, At_{J-1}^{nd_x^0}\}$. Node nd_y^1 has attribute list $AtList - nd_y^1 = \{At_0^{nd_y^1}, At_1^{nd_y^1}, \ldots, At_{J-1}^{nd_y^1}\}$, such that $At_0^{nd_x^0} = At_0^{nd_y^1}$, $At_1^{nd_x^0} = At_1^{nd_y^1}, \ldots, At_{J-1}^{nd_x^0} = At_{J-1}^{nd_y^1}$.
- Condition3: The input for node nd_x^0 is data sources list $DsList - nd_x^0 = \{Ds_0^{nd_x^0}, Ds_1^{nd_x^0}, \ldots, Ds_{K-1}^{nd_x^0}\}$ and the input for node nd_y^1 is data sources list $DsList - nd_y^1 = \{Ds_0^{nd_y^1}, Ds_1^{nd_y^1}, \ldots, Ds_{K-1}^{nd_y^1}\}$ such that $Ds_0^{nd_x^0} = Ds_0^{nd_y^1}$, $Ds_1^{nd_x^0} = Ds_1^{nd_y^1}, \ldots, Ds_{K-1}^{nd_x^0} = Ds_{K-1}^{nd_y^1}$.

Condition1 and condition2 guarantee that mashups Mp_0 and Mp_1 contain the same node while condition 3 makes sure that nodes nd_x^0 and nd_y^1 have the same data sources as input. A list of consecutive nodes in mashup Mp_0 $CoList - Mp_0$=$\{nd_f^0, nd_{f+1}^0, \ldots, nd_{H-1}^0\}$ and a list of consecutive nodes in mashup Mp_1 $CoList - Mp_1$=$\{nd_z^1, nd_{z+1}^1, \ldots, nd_{R-1}^1\}$ are considered common if and only if:

- Condition4: nd_f^0 *COMMON* nd_z^1, nd_{f+1}^0 *COMMON* $nd_{z+1}^1, \ldots, nd_{H-1}^0$ *COMMON* nd_{R-1}^1.
- Condition5: nodes forming $CoList - Mp_0$ appear in the same order as nodes forming $CoList - Mp_1$.

4.2.2 Common Component Detection Technique (CCD)

This section describes our technique for detecting common components. The technique is targeted towards finding the longest common components sequences across mashups. The

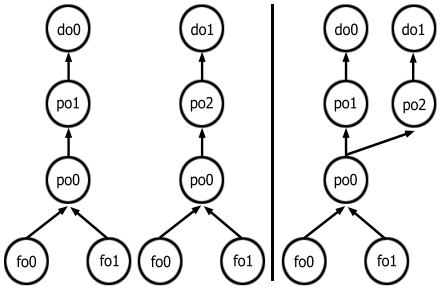

Figure 4.2: Two mashups and the result of merging them

longer the sequence, the more components are considered as common across mashups which leads to more savings. To the best of our knowledge, subtree matching algorithms focus on matching subtrees of a pair (or a small set) of trees. However, they do not consider situations where a system would contain large numbers of trees and subtrees. This raises significant efficiency and scalability issues. Our system consists of potentially large number of mashups because of the personalization property of Web 2.0. Therefore, we needed an algorithm that efficiently indexes mashups and utilizes the index for scalable subtree matching.

The input for our algorithm is a set of mashups $MpSet = \{Mp_0, Mp_1, \ldots, Mp_{N-1}\}$ and the output is a structure of merged mashup trees. This structure reflects mashups taking into consideration common components. Figure 4.2 is an example of two mashups and the resulting structure of merging them. Each mashup consists of a set of branches such that each branch Br_i starts with a single fetch operator and ends with the single dispatch operator. Figure 4.3 shows a flowchart describing the common component detection process. The algorithm starts by iterating through all mashups in the system. For every mashup, we start at the beginning of each branch of the mashup. In other words, we start at the fetch operator in every branch of the mashup tree, if the operator's representation exists in the components index, then we move to the next operator (its parent) to check its existence in the components index, we keep going for the next operator, because we are looking for the longest sequence

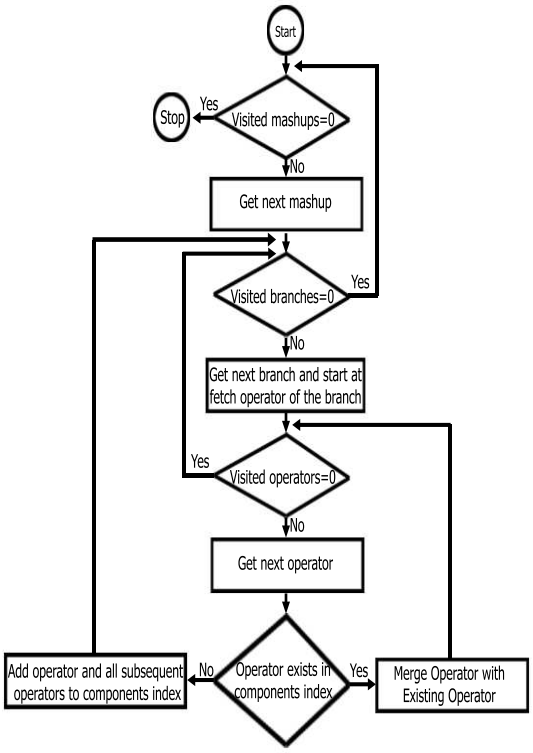

Figure 4.3: Common Component Detection Description

of operators shared across mashups. If the operator's representation does not exist in the components index, this means that all subsequent operators in the same branch also do not exist in the index because as explained in Section 3.2 each operator's representation includes the representation of its children. As a result, the operator and all subsequent operators in the mashup are added to the components index. If no further operators are left in the current mashup branch, then we move to the next mashup branch and when all operators within all branches of a single mashup are visited, then we move to the next mashup. The algorithm stops after iterating over all mashups in mashup set $MpSet$.

The common component detection process is efficient because each mashup is merged with existing mashups as the mashup is requested by end-user. The efficiency of common component detection also comes from using the components index described in Section 3.2 which

makes accessing mashup components more efficient process. For the sake of generality, we described the input of the algorithm as a set of mashups to be merged together. However, mashups are merged on the fly as they are requested by end-users.

To analyze the complexity of our algorithm, consider the case where a new mashup enters the system and the number of operators in the mashup tree is m. We need to traverse all operators in the mashup tree to find out if they exist in our components index. In the worst case, none of the mashup operators exist in the system. Traversing the mashup tree is of order $O(m)$. Now, in each traversed operator, we search the B+ tree index once for that operator, searching B+ tree complexity is $O(t * log_t n)$, where t is the degree of the B+ tree and n is the number of keys in the B+ tree. As a result, the whole algorithm complexity is $O(m * t * log_t n)$ which is of a quadratic complexity.

4.3 Operator Reordering

The objectives of reordering are two fold; (1) Standardize the internal representation of mashups; two mashups that operate on the same data sources, use same set of operators and yield same end-results have identical representations. This improves the effectiveness of common component detection because it increases the probability of detecting common components; (2) Transform the mashup into a form that is more efficient from performance standpoint. For example, executing a filter operation before a sort operation is more efficient than the reverse order of execution.

4.3.1 Commutable Operators

One approach for standardizing mashup representation is to restrict a specific order on mashup operators such that the semantics of the mashup does not change. This can be achieved by detecting operators that can be interchanged without modifying mashup end-result; these operators are called commutable operators. The following operators are considered to be commutable

- Sort, filter, reverse, and unique are commutable
- Sub-element and sort are commutable; only when both operate on same attribute
- Sub-element and filter are commutable only when both operate on same attribute
- Sub-element and reverse are commutable

Based on the previous lists of commutable operators, a special order of commutable operators can be set which is described in the next subsection.

4.3.2 Canonical Form

Our canonical form defines a set of rules that we apply on mashups to make their design more standard. These rules are analogous to optimizing relational algebra expression trees. Our canonical form has the following rules. In these rules x → y means that operator x precedes operator y:

1. Filter → Union
2. Filter → Sort → Reverse → Unique
3. Sub-element → Sort
4. Filter → Sub-element
5. Sub-element → Reverse
6. In any union operator, data sources appear in their lexicographical order.

If a union operator merges contents of data sources sports.yahoo.com and news.google.com, then news.google.com should appear to the left of sports.yahoo.com after enforcing the canonical form. The reason for making filter operators appear first in the previous rules is that a filter operator usually refines the data it receives as an input and the resulting output would usually be less in size than the input, this makes the operators that follow the filter operator execute faster because the size of their input data becomes smaller.

Figure 4.4 shows an example of a mashup designed by an end-user and its design after enforcing the canonical form. Canonical form rules are followed to make sure that all mashups

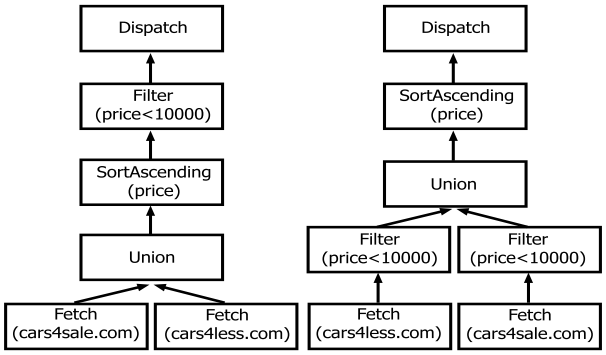

Figure 4.4: A mashup and its conversion to canonical form

abide to a more strict design structure. By applying the canonical form, we increase the probability of detecting common components across mashups and at the same time improve the efficiency of executing mashups. Notice that we cannot rely upon end-users to enforce canonical form, as they may not have the required expertise. After the end-user completes designing his mashup, the system enforces the canonical form rules on the mashup.

In this part, we proposed common component detection and canonical form to enhance the execution of mashups. This is beneficial in cases where data sources involved in mashups are continuous (eg. streams). In the second piece of our work, we consider the case when data sources involved in mashups are mostly static or do not change a lot. Towards addressing this issue, in the next chapter, we propose caching as a mechanism of improving the efficiency of mashup platforms.

Chapter 5

Caching for Mashups

This chapter explores caching as a mechanism to alleviate the scalability challenges of mashups. Caching is a proven strategy to boost performance and scalability of Web applications. For example, Web content delivery and Web services have long adopted caching [99]. Several caching techniques have been specifically developed for Web services [79, 80]. However, most of these techniques cannot be directly used for mashups. Most Web service caches store the final results of Web service workflows or at pre-specified stages of the Web service processes[1]. We contend that this *static* caching strategy would not be effective for mashups due to the inherent differences between mashups and Web service processes. Therefore, investigating caching techniques for mashups is important, especially in the cases where data sources do not change quite often.

As Web services are authored by professional developers, it is possible for them to identify the stages of Web services at which the results should be cached. On the other hand, mashups are created by individuals with varying degrees of technical expertise, so caching decisions are left to the mashup platform. Further, data sources for Web services are mostly hosted on the same machine hosting the Web service itself and that makes caching decision easier. On the other hand, mashups require data from external sources; mashup platforms have no control on these data sources.

Because of these traits, mashups demand a more dynamic caching strategy, wherein: **(a)** the intermediate results of mashup computations can be stored for future use; **(b)** the cache enables the intermediate results of one mashup to be used in another; and **(c)** the caching

[1]The cache is explicitly configured to store results at a certain point of the workflow.

decisions are based upon dynamic benefit-cost analysis which takes into account the structural characteristics of mashups but is also adaptive to the various dynamics of the mashup platform.

This chapter describes the design and evaluation of *MACE* (**ma**shup **c**ach**e**) - a server-side cache framework for the mashup domain. The design of the MACE framework embodies three original contributions.

- We design a mashup structure-aware scheme for indexing cached data which enables MACE to efficiently discover whether any of the currently cached data can be reused in the execution of a newly created mashup.
- We incorporate taxonomy-awareness and provide support for range queries to further increase reuse of cached data.
- We present a dynamic cache point selection scheme that estimates the benefits and costs of caching data at different stages of mashup trees. Our approach selects a set of points that collectively maximize the benefit-to-cost ratio of caching data at those points.

The previous contributions are introduced in MACE for the purpose of increasing the efficiency of mashup platforms.

5.1 MACE Architecture

Figure 5.1 illustrates the architecture of the MACE system. The MACE system is co-located with the mashup platform. However, it is designed such that the mashup platform itself would require minimal modifications to work in conjunction with MACE.

In order for MACE to select stages at which data will be cached, it continuously observes the execution of mashups, and collects statistics such as request frequencies, update rates and cost and output size values at various nodes of the mashups. It then performs cost-benefit analysis of caching at different nodes of mashups, and chooses a set of nodes that

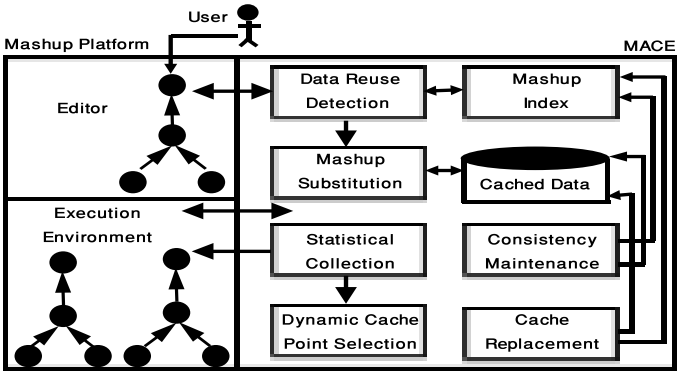

Figure 5.1: MACE Architecture

are estimated to yield best benefit-cost ratios. An operator node in a mashup tree that is chosen for caching by MACE (i.e., the results until that stage of the mashup execution would be stored) is called a *cache point*. Any node in the mashup tree except the root of the tree (corresponding to the dispatch operator) can potentially be chosen as a cache point. This set of nodes is called the *potential cache point set (PcpSet)*, and individual nodes in this set are referred to as *potential cache points*.

MACE also interacts with the mashup editor to obtain newly created mashups. For each new mashup, the MACE platform analyzes whether any of the cached results can be substituted for part of the mashup workflow. If so, the mashup is modified so that cached data is re-used, and only the additional operations required for completing the mashup are performed. The modified mashup is then provided to the mashup platform for execution. In addition to these two main features, the MACE platform also incorporates the basic cache functionalities such as replacement scheme and data consistency mechanism. This chapter focuses on the design of a dynamic cache point determination technique and mechanism to re-use the cached data for substituting parts of incoming mashups. The next sections describe these two unique features of the MACE platform.

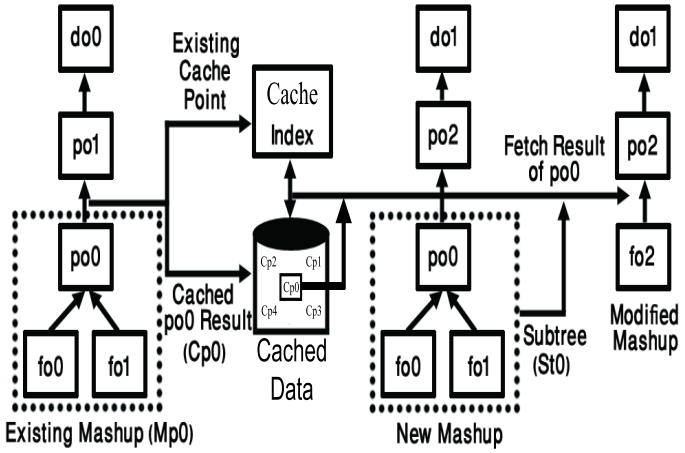

Figure 5.2: Cached Data Reuse in MACE

5.1.1 Cache Indexing for Efficient Data Reuse

Determining points of data reuse in new coming mashups is not a straightforward task. Notice that a cache point represents the results of computations occurring in a *subtree* of the mashup tree. This subtree itself might have one or more branches with fetch operators at the leaves. The results at a particular cache point Cp_h can be reused for an incoming mashup Mp_l if and only if the subtree represented by Cp_h *exactly matches* a subtree in Mp_l. By exact matching, we mean that a subtree of the incoming mashup has the same structure as that of the subtree represented by Cp_h, and parameters of operators in both subtrees are the same. Since mashup platforms support large numbers of mashups, we need a scalable mechanism to find out whether one or more subtrees of a new mashup match existing cache points. MACE includes a novel cache point indexing scheme to address this issue, which is explained later in this section.

If one or more subtrees of a new mashup Mp_l are found to match existing cache points in the MACE system, Mp_l is modified as follows. For each subtree that matches an existing cache point, the subtree is replaced with a fetch operator that references the cached data corresponding to the cache point. For example, if an arbitrary subtree St_q of a Mp_l, matches

an existing cache point Cp_h, St_q is replaced with a fetch operator that refers to the cached data corresponding to Cp_h. The modified mashup is then sent to the mashup platform which executes it. Figure 5.2 illustrates the modification of a new mashup to reuse data available in the cache. We now explain our mashups representation and indexing scheme that enables efficient discovery of cache points.

5.1.2 B+ tree Cache Index

Section 5.1.1 described a B+ index that facilitates accessing mashup components. Another version of this index is used to facilitate searching for cache points. The index nodes' entries of the new version of the B+ tree are substrings of mashups detailed string representations. Those substrings correspond to representations of cache points. They are entered to the index based on their lexicographical order. In other words, the index in Section 5.1.1 is used for indexing mashup components (operators), while the new version we use in this chapter is used for indexing cache points. Consider the following mashup example, Fetch data source buycars.cars.com, filter data based on model=Honda, sort on price, if we decide to cache after the filter operator is executed, then 12#15|12|04|30|$Honda$ will be inserted into the index. But if we decide to cache after the whole mashup execution flow is done, then 12#15|12|04|30|$Honda$#09|12|06 is inserted into the index.

Figure 5.3 shows the cache index if we decided to cache after both of the previous 2 points. The numbers in identification strings are IDs of the data sources, operators and attributes forming a mashup, for example, the filter operation 15|12|04|30|$Honda$ is interpreted as follows, 15 is the ID of the filter operator, 12 is the data source ID from which attribute 04 is taken, 30 is the ID of the equality operator and $Honda$ is the value on which the attribute 04 is filtered. In the previous identification strings # represents a special character which works as a separator between operators. Similar to the index described in Section 5.1.1, in the new index version we use here, each operators' detailed string representation reflects the operators which precede it in the mashup workflow.

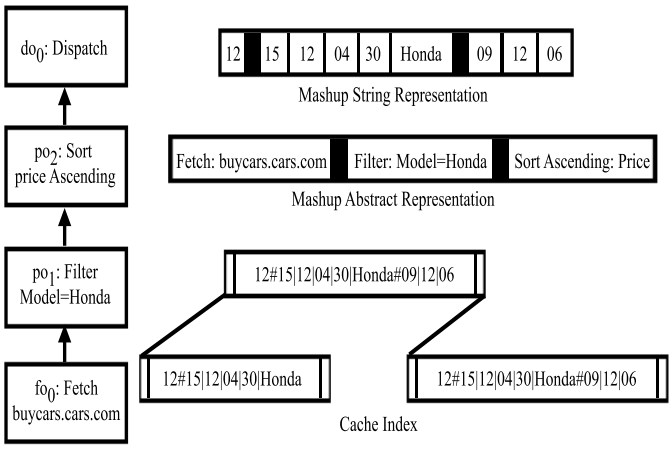

Figure 5.3: Mashup representation and cache index

5.2 Range Queries and Taxonomy Awareness

Until now, our design of the indexing scheme is focused on lookups for exact matches. However, supporting lookups for inexact matches can considerably increase the reuse of cached data. Consider the case when the cache contains superset of data needed for an incoming mashup. In this case, the data in the cache can be appropriately filtered and reused for the new mashup. Unfortunately, looking up for exact matches would fail to even locate the existence of the superset, thereby precluding the possibility of data reuse. Towards addressing this issue, we enhance our indexing mechanism to support two specific kinds of inexact matching, namely, range queries and hierarchical taxonomies. Our strategy relies on the fact that the keys in the leaf level of the B+ tree index are sorted, which implies that mashups that are lexicographically close by to one another are stored either in the same index node or in a nearby node.

5.2.1 Supporting Range Queries

Suppose that a new mashup requires data with parameter p being in the range interval $[x, y]$. With exact matching, if the cache index does not contain the precise range $[x, y]$, a cache miss occurs and the new mashup is executed end-to-end. Consider the scenario when the cache contains the results of an earlier mashups that is similar to the new one except that parameter p is in the range $[a, b]$. Now the question is whether this data can be reused for the new mashup? In order to determine this, we need to consider three distinct cases.

First, if $x = a$ and $y = b$, then this means that range interval $[x, y]$ is fully included within range interval $[a, b]$ which also means that the result of the new mashup is fully contained within the result of the existing mashup. In this case, a cache hit is declared and a local search within the result of the existing mashup is performed to extract the result of the new mashup. Second, if $x = a$ and $y > b$ or $x < a$ and $y = b$ or $x < a$ and $y > b$, then the range interval $[x, y]$ is partially included in the range interval $[a, b]$. Here, because the result of the new mashup cannot be completely satisfied by the existing cached mashup result, then a cache miss is declared and the new mashup is executed end-to-end. It is noteworthy that this partial inclusion relationship can be useful in two situations. First, the case where partial results can be extracted from cached data and we query for missing data, then the missing data and partial results are combined. Second, the case where the new mashup cannot be executed due to difficulties in communicating with data sources. In such a situation, the data available in the cache is reused to provide the end-user with a valid but incomplete result. For example, if we consider the $x = a$ and $y > b$ scenario, the result satisfying the range interval $[x, b]$ can be provided for the end-user from cached data. Such a result is not complete, but it is still valid and may be useful to the end-user. Third, if $x < a$ and $y < a$ or $x > b$ and $y > b$, then the range $[x, y]$ is totally outside the range $[a, b]$. In this case, a cache miss is declared and the new mashup is executed end-to-end.

The inherent capability of the B+ tree structure to lookup range values can be leveraged for the above purpose. The following example illustrates how this is achieved in the MACE

framework. Suppose the following mashup result is cached in the system: Fetch data from buycars.cars.com then filter data based on price < 5000. Based on our mashup string representation, this mashup is represented in the cache as 12#15|12|06|32|5000 where the last 4 digits correspond to the value on which data is filtered (5000). Now, suppose an end-user asks for a new mashup which is described as follows, fetch data from buycars.cars.com then filter data based on price < 4000, the new mashup is represented as 12#15|12|06|32|4000. This case represents case 1 where the range interval of the new mashup is fully included within the range interval of the existing mashup. We can see that the two mashups have some common part (12#15|12|06|32) in their string representation. Without using range query improvement, a search for the new mashup representation 12#15|12|06|32|4000 results in a cache miss, this happens because we make exact matching between index keys and the mashup representation we are looking for. When using range query awareness, the search process for previous mashup (12#15|12|06|32|4000) in the tree explores through index levels based on lexicographical order of keys and eventually arrives at the existing mashup key 12#15|12|06|32|5000. Now, instead of declaring a cache miss, we detect that this key (12#15|12|06|32|5000) and the key we are looking for (12#15|12|06|32|4000) represent the same mashup except that the value on which data is filtered is different. Here, we do not have to execute the new mashup right from its starting point, instead, we search items that the previous key points to and then exclude items with price between 4000 and 5000.

Accordingly, we achieve better utilization out of cache index. As an example of case 2, suppose the new mashup is filtering data based on price < 6000, here the result of the new mashup is partially included in the existing mashup cached result. Normally, we declare a cache miss and execute the new mashup from scratch, but if the new mashup execution is interrupted due to communication problems, then a cache hit is declared and the end-user is provided with the result of car items cheaper than 5000. One might argue that providing incomplete result is not accurate. Although this is true, the partial result can satisfy end-user demands in many cases, here, the end-user might find a suitable car for him.

Note that when MACE caches mashup results, it may so happen that two results sets (of two distinct mashups) overlap without anyone of them being a subset of the other. In such a scenario, we do not make duplicate copies of data that is common to both sets. We maintain a single copy of the common items but store the pointer at two distinct locations in the index. This maximizes the utility of the available storage.

5.2.2 Supporting Hierarchical Taxonomies

The range query technique presented in the previous section works well for numeric parameters. However, in many cases, end-users can create mashups that extract general information, while others might create mashups that extract more specific information with respect to parameters that are non-numeric. Consider the case when the mashup platform caches the results of a mashup that extracts all sports related stories from sports.yahoo.com. Now, suppose another end-user creates a mashup that extracts all stories related to Tennis from sports.yahoo.com. Clearly, the results of the new mashup are more specific and constitute a subset of the results of the existing mashup. The previous range query mechanism cannot be used for this case as the parameter is keyword-based.

We have developed a mechanism to detect these types of generic/specific relationships among mashups in terms of keyword parameters. The central idea is to build a hierarchical taxonomy that defines relationship between various keywords or categories. The assumption is that a keyword at a higher level in the hierarchy subsumes all keywords which reside underneath it. This hierarchical taxonomy is used to enhance data reuse and minimize cache misses. When a new mashup is created, our strategy is not to just search the index for earlier mashups with exactly matching keywords, but also to look for existing mashups that have ancestors of the keywords in the incoming mashup. Specifically, suppose an existing mashup filters data based on non-numeric keyword-valued parameter p being equal to X, and suppose this result is cached. Later, suppose a new mashup which is identical to the existing mashup except that p equals Y is created. Normally, a cache miss is declared and the new mashup is

executed end-to-end. However, providing that a hierarchical taxonomy exists, we can use it to look for a possible relationship between X and Y. If X is an ancestor (direct or indirect parent) of Y, then the result of new mashup is a subset of the cached result. However, the cached results cannot be directly used for the new mashup. The cached results have to be locally filtered to the actual result of the new mashup.

As an example, suppose the cache contains the result of the mashup that fetches data from sports.yahoo.com, then filter data based on the criterion category = sport. The index has the key 10#15|10|02|30|*sport* corresponding to this data in the cache, where sport is the value on which data is filtered. Now, suppose an end-user creates a mashup to fetch data from sports.yahoo.com, then filter data based on category = Tennis. The key for the new mashup is 10#15|10|02|30|*tennis*, where Tennis is the value on which data is filtered. The search process for the new mashup in the cache starts by exploring the index until we reach the level containing the key 10#15|10|02|30|*sport*. If we are going to use exact matching to look for the new mashup in the cache, we will end up with a cache miss. Instead, we detect that the part 10#15|10|02|30 is common between the new mashup and the cached mashup, so we extract the value on which the cached mashup is filtered (sport) and we extract the value on which the new mashup is filtered (Tennis), then we consult the taxonomy to look for a possible relationship between these two keywords. Since Tennis is a child of sport in the taxonomy, we conclude that the result of the new mashup is contained in the result of the previously cached mashup. Consequently, the result of the new mashup can be found by locally filtering the cached data of the first mashup. Figure 5.4 illustrates the above example.

5.3 Dynamic Cache Point Selection

In this section, we describe our dynamic cache point selection technique. We formulate the dynamic cache point selection as an optimization problem following which we provide efficient algorithms for cache point selection.

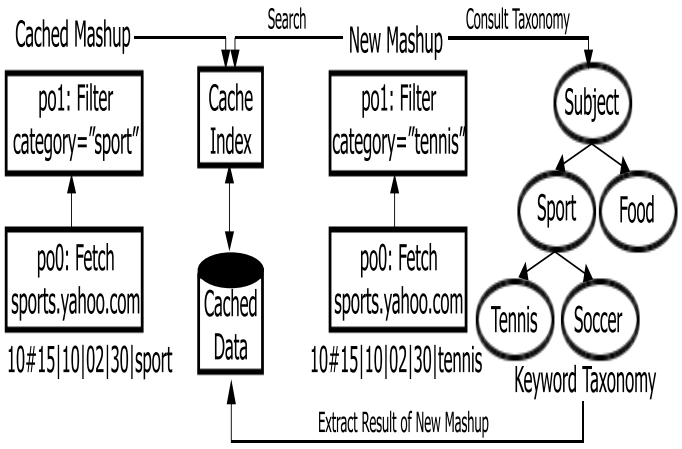

Figure 5.4: Taxonomy Support in MACE

5.3.1 Problem Formulation

This section formulates the cache point selection as a cost-benefit optimization problem. We provide two flavors of the cost-benefit optimization problem. The first one models a scenario wherein the storage-space availability at MACE is unlimited and the second corresponds to the scenario in which the MACE system has limited storage capacity. We begin by introducing terminologies and notations that are employed in the problem formulation.

Potential cache point set ($PcpSet = \{Pcp_0, Pcp_1, \ldots, Pcp_{M-1}\}$) represents the *unique* potential cache points corresponding to the mashups existing in the $MpSet$. Recall that every operator node in a mashup except the root is a potential cache point. The members of $PcpSet$ are unique in the sense that the potential cache points that represents subtrees which exist in multiple mashups are included only once. The sum of the cost values of all the descendant nodes of a potential cache point Pcp_k including the cost value of Pcp_k is called the *cumulative cost value* of Pcp_k ($CCV^{Pcp_k} = CV^{Pcp_k} + \sum_{Pcp_h \in Descendent(Pcp_k)}(CV^{Pcp_h})$).

5.3.2 Cost-Benefit Analysis

The benefits of caching the results at a particular potential cache point Pcp_k is that the cached data would be re-used for any future requests of all mashups that Pcp_k is part of, thus avoiding the re-executions of Pcp_k and all of its descendant nodes. Let *request frequency* of Pcp_k (represented as RF^{Pcp_k}) denote the number of times Pcp_k needs to be executed per unit time to satisfy user requests if the output of Pcp_k is not cached. Note that RF^{Pcp_k} is the total sum of the request frequencies of all the individual mashups that the subtree under Pcp_k is part of. Thus, the benefits per unit time obtained by caching at Pcp_k is $RF^{Pcp_k} \times CCV^{Pcp_k}$.

Caching at a potential cache point Pcp_k involves two distinct costs, namely *consistency costs* and *storage costs*. Consistency costs are the costs involved in maintaining the consistency of cached data in the face of updates to the data from external sources that are used in computing the output Pcp_k. Notice that the data cached at Pcp_k becomes invalid, and would need to be updated anytime the data obtained through any of the fetch operators below Pcp_k changes. Each time the output of Pcp_k needs to be re-computed, Pcp_k and all of its descendant nodes need to be re-executed. Thus, the consistency costs per unit time of caching at Pcp_k can be quantified as $UF^{Pcp_k} \times CCV^{Pcp_k}$, where UF^{Pcp_k} represents the sum of the update frequencies of all the external data sources fetched by the operators below Pcp_k.

The storage costs of caching at Pcp_k is directly proportional to the size of the output (OSV^{Pcp_k}). However, notice that the storage costs only matter when available storage is limited. Furthermore, storage costs and consistency costs are inherently different, and cannot be combined into a single equation in a meaningful way. We model the storage costs as constraint rather than optimization criterion.

$RF^{Pcp_k} \times CCV^{Pcp_k} - UF^{Pcp_k} \times CCV^{Pcp_k}$ is called the *cost-benefit trade-off* for Pcp_k (represented as CBT^{Pcp_k}). CBT^{Pcp_k} quantifies the net cost-savings obtained by caching at Pcp_k. Note that in this formulation of CBT^{Pcp_k}, the computational overheads incurred at the time of serving user requests and those incurred to maintain consistency of cached data, are of

equal importance. In scenarios where one is more important than the other, the two terms of CBT^{Pcp_k} have to be appropriately weighted to reflect their relative importance.

Scenario 1 −− No storage limitations: As stated earlier, the objective of the dynamic cache point selection scheme is to select a set of cache points such that the benefit-cost tradeoff is maximized. Let X^{Pcp_k} be a $\{0, 1\}$ variable denoting whether Pcp_k is selected as a cache point (X^{Pcp_k} is 1 if Pcp_k is chosen and 0 otherwise). Therefore, the optimization criterion would be to assign X^{Pcp_k} values to each potential cache point $Pcp_k \in PcpSet$ such that $\sum_{Pcp_k \in PcpSet} X^{Pcp_k} \times CBT^{Pcp_k}$ is *maximized*. However, notice that the optimization problem, as it stands, can lead to *duplicate-caching* (caching same or interdependent data multiple times thus wasting resources). In order to avoid this, we introduce the following constraint. For any pair of potential cache points $\{Pcp_k, Pcp_i\}$ such that $Pcp_k \in Descendant(Pcp_i)$ or vice-versa, $X^{Pcp_k} + X^{Pcp_i} \leq 1$.

Scenario 2 −− Limited storage The optimization problem formulation for the limited storage scenario is similar to the previous case, but the total storage requirements of cached data should not exceed the storage available in the MACE system. Suppose Sg denote amount of storage available. The optimization problem can be stated as follows. Assign values to decision variables $\{X^{Pcp_0}, X^{Pcp_1}, \ldots, X^{Pcp_{(M-1)}}\}$ corresponding to the potential cache points $\{Pcp_0, Pcp_1, \ldots, Pcp_{M-1}\}$ such that $\sum_{Pcp_k \in PcpSet} X^{Pcp_k} \times CBT^{Pcp_k}$ is *maximized* while ensuring that the following constraints are not violated: (1) $X^{Pcp_k} \in \{0, 1\}, \forall Pcp_k \in PcSet$; (2) $\forall \{Pcp_k, Pcp_i\}$ such that $Pcp_k \in Descendant(Pcp_i) || Pcp_i \in Descendant(Pcp_k), X^{Pcp_k} + X^{Pcp_i} \leq 1$; and (3) $\sum_{Pcp_k \in PcSet} X^{Pcp_k} \times OSF^{Pcp_k}(ips) \leq Sg$, where the variable ips represents the inputs to the operator at Pcp_k as specified in the mashups. Notice that this is a constrained discrete optimization problem solving which requires exhaustive search of the solution space. In the next section, we present a greedy strategy-based algorithm for this problem.

5.3.3 Cache Point Selection Algorithms

First, we consider the scenario wherein storage space is not a constraint. Statistics such as the request and update frequencies of potential cache points are collected, and the corresponding cumulative cost values are calculated. For each mashup in the platform, our algorithm searches for the best cache point as follows. The algorithm starts searching from the potential cache point that is shared across many mashups, and at the same time is located at lower-levels of the mashup tree. This can be achieved by starting at a node that has the maximum value for $\frac{SMCount}{Height}$, where $SMCount$ (sharing mashups count) indicates the number of mashups that share the potential cache point and $Height$ indicates its height in the mashup. The rationale for starting the search at such a node is that it is likely to yield maximum reuse (thereby maximizing the benefits) at low consistency maintenance costs. Suppose the algorithm starts from the potential cache point Pcp_k. The node that is currently being searched is called the *current search point (CSP)*. We calculate CBT^{CSP} as $RF^{CSP} \times CCV^{CSP} - UF^{CSP} \times CCV^{CSP}$. We then compare the value of CBT^{CSP} to the CBT value of its ancestor in the mashup and the CBT value of its descendant in the mashup (if Pcp_k has multiple descendants, we consider the sum of their CBT values). If the CBT value of the ancestor is higher than that of Pcp_k, the ancestor is initialized as the new CSP, and the algorithm continues searching upwards from that point. If, on the other hand, the descendant node had a higher CBT value, the descendant is initialized as the new CSP and the algorithm continues searching downwards. If Pcp_k has multiple descendants, the algorithm continues searching downwards from each of them.

The search terminates when we reach one or more nodes such that the CBT values of their respective descendants and ancestors are lower than their CBT values.[2]. The potential cache point(s) at which the search terminates are chosen to be included in the *cache point set (CPSet)*. This algorithm yields optimal solution to the scenario with no storage limitations. The algorithm is linear in terms of the number of potential cache points in the platform.

[2]The search may also terminate when we reach the end of the mashup tree (in either direction)

We now extend the above algorithm for the limited storage scenario. Recall that discovering optimal solutions for this scenario requires exhaustive search of the solution space. Therefore, our objective is to design an efficient algorithm that yields close to optimal solutions. The algorithm for the limited storage scenario works in two stages. The first stage is exactly similar to the algorithm described above for the scenario wherein the storage space is not a limitation. However, the storage requirements for $CPSet$ obtained at this step may exceed the available storage (Sg). The second stage of the algorithm performs additional level of pruning as follows. For each cache point Pcp_k in the CPSet produced at the end of first step, it calculates the ratio $BCS^{Pcp_k} = \frac{CBT^{Pcp_k}}{OSV^{Pcp_k}}$. This ratio quantifies the per-byte cost savings obtained by caching the results of Pcp_k. The cache points are sorted in the descending order of their BCS values. The algorithm then progressively eliminates the cache points from the end of this sorted list (i.e., the cache points with the least BCS values are eliminated first) until the results of the cache points remaining in the $CPSet$ can fit into the available storage. The rationale for this elimination strategy is to retain cache points that provide maximum benefits for the amount of storage space they consume. Once the $CPSet$ is computed, the MACE engine starts storing the outputs of the cache points.

In this chapter and the previous one we proposed two key pieces for improving the efficiency and scalability of mashup platforms; 1) We proposed common component detection and operators reordering to improve the execution of mashups which use continuous data sources. 2) We proposed a caching architecture and a caching protocol for enhancing execution of mashups which use static or less frequently changed data sources. To the best of our knowledge, current mashup platforms are based on centralized servers which degrades their scalability and makes them vulnerable because of failures. In the next chapter, we target this issue by proposing a distributed mashup architecture that is scalable and failure resilient.

Chapter 6

Distributed Mashup Platform

This chapter explores distribution as a mechanism to achieve scalability and performance of mashups. To the best of our knowledge, most of existing mashup platforms are centralized; this results in several drawbacks. First, since mashup platforms typically host large numbers of mashups and experience high mashup request rates, a centralized mashup platform faces an increasing pressure and might not be able to keep up with increasing amount of end-users requests which raises a scalability problem. Second, a centralized mashup platform does not consider the geographical location of end-users and data sources; this implies that some end-users might observe high delays. Third, having a centralized mashup platform implies that the centralized server is a single point of failure. The previous three points motivates the need for a distributed mashup platform where mashup execution takes part on several cooperative nodes.

Distributing mashup execution requires the collaboration of distributed nodes in an overlay that faces network dynamics; therefore, we need to handle several challenges in distributing mashup execution. Since we are designing a distributed system, what type of cooperation is needed between network nodes? This is important to guarantee a complete and correct mashup execution. Also, should a mashup be executed on one node or multiple nodes? Executing a mashup on multiple nodes forms a way of parallel execution. Further, what are the parameters based on which a node is selected to execute the whole mashup or part of it? Parameters such as communication links delay, bandwidth and nodes loading should be considered.

The next challenges are related to handling dynamics of the system; what happens in the case of changing network parameters? For example, communication links delay and bandwidth between nodes are changing due to factors such as congestion. Also, node loading is changing as nodes get more mashups to execute. Therefore, a mashup that is being executed on some nodes might have to be reassigned to different nodes to adapt to changing network parameters. Also, what if more than one end-user share the same mashup or part of it? A mashup execution plan is initially designed based on a number of end-users requesting it. When more end-users request the same mashup, the mashup execution plan might change based on the geographical location of the new end-users. Further, what happens if network nodes fail? Certain recovery method should be applied to make sure system functionality is not affected?

Towards addressing the previous challenges resulting from having a centralized mashup platforms, we design a cooperative overlay-based architecture called *CoMaP* which considers distribution as a mechanism to enhance the scalability of mashup platforms. In *CoMaP*, processing nodes cooperate to execute mashup workflows and a mashup can be executed on one or more processing node. In designing *CoMaP*, we make three novel contributions.

- First, we present an efficient cooperative mashup architecture in which multiple nodes cooperate to execute mashups. Our architecture is distributed and mashup operators are spread across several nodes. The collaboration between mashup execution nodes is facilitated by a controller which also plans the execution of individual mashups. We formally model the system and the components which interact with the system. We also formally define costs involved in executing mashups in a distributed fashion and we formally define the distributed mashup execution problem as an optimization problem.

- Second, we introduce a dynamic mashup distribution technique that is sensitive to the locations of the various data sources of an individual mashup as well as the destinations of its results. Our technique progressively optimizes the network load in the overlay and the latency of mashup execution. Our technique depends on a two-stage optimization

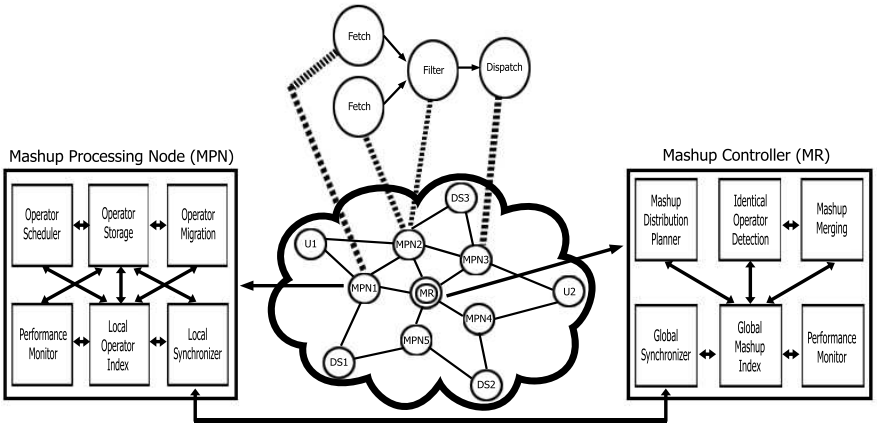

Figure 6.1: CoMaP Architecture

process. First, a cheap cost first stage is used to reach a good distribution decision. Then, a second stage which is executed periodically is used to reach a close to optimal distribution decision.

- Third, we handle failure resiliency issues in our architecture through replicating nodes and replicating parts of mashup workflows. We explain the type of communication and interaction necessary between overlay nodes to support failure resiliency and we show the impact of failure resiliency on the system.

We simulated a distributed architecture for mashup execution for the purpose of applying our techniques in this simulated environment.

6.1 CoMaP Architecture

Figure 6.1 illustrates *CoMaP*'s high-level design architecture. *CoMaP* is based upon an overlay of mashup processing nodes MPNs, we refer to this set of nodes as $MPNSet = \{MPN_0, MPN_1, \ldots, MPN_{L-1}\}$. These nodes are distributed across the Internet, and they collaborate to execute mashup workflows. A mashup controller MR plans the execution of

each mashup. It also coordinates the activities of various processing nodes involved in the execution of a mashup. A set of end-users $USet = \{U_0, U_1, \ldots, U_{Z-1}\}$ interact with $CoMaP$. Each of those end-users can design and submit his own mashups to MR using a mashup designer. Note that each MPN and end-user in $CoMaP$ can tell what its coordinates are by probing a set of nodes geographically distributed over the Web (Landmarks). Landmarks [100] cooperate among each other to deliver coordinates for the node that probed them. We refer to the set of mashups that MR receives as $MpSet = \{Mp_0, Mp_1, \ldots, Mp_{N-1}\}$. Operators that form those mashups are referred to as $OpSet = \{Op_0, Op_1, \ldots, Op_{M-1}\}$. The previous entities are connected with a set of communication links where each communication link $LNK_{e,f}$ connects node e with node f. Each communication link $LNK_{e,f}$ has a delay $DeLNK_{e,f}$ and a bandwidth $BndLNK_{e,f}$.

Each MPN is responsible for executing a set of workflows as determined by MR. An individual workflow might correspond to an entire mashup or part of it. A mashup workflow is essentially a tree of operators. When executing a workflow, one MPN may fetch data from external sources or it may receive partially processed data from other processing nodes which would have executed earlier parts of the mashup. The results are dispatched either to the end-user (if no further processing is needed for the mashup) or to another MPN (if mashup execution is not yet complete). Executing a mashup on several nodes yields better efficiency and scalability. For example, if a mashup consists of two fetch operators that fetch data from two different data sources, then assigning each operator to a different node facilitates parallel execution.

As indicated in Figure 6.1, each MPN comprises of several components. Mashup workflows assigned to the processing node are stored in the operator storage, and are indexed by the operator index. The local scheduler makes the workflow scheduling decisions. The performance of each processing node is locally monitored, summary of which is communicated to the global performance monitor (located at MR) periodically.

End-users interact with the system through MR which they get to know upon joining the

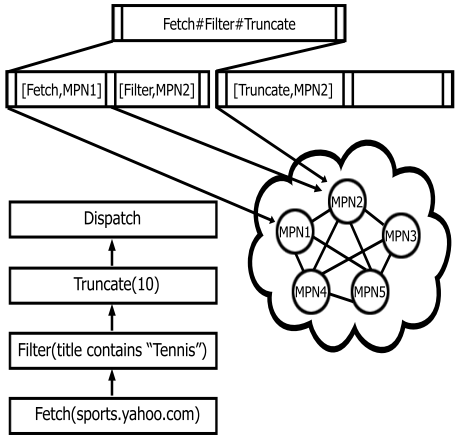

Figure 6.2: A mashup stored in B+ tree which points to MPNs where operators are deployed

system. The set of interactions include creation and deployment of new mashups and deletion of existing ones. When an end-user sends a new mashup to MR to be executed, MR first checks whether the newly created mashup shares operator sequences with existing mashups. If so, MR modifies the mashup to utilize the results from the existing operator sequences so that duplicate computations are avoided. The global mashup index aids the detection of shared mashup operators. This index is similar to the index that we introduced in *AMMORE* (Chapter 4) except that the index introduced in *AMMORE* is based on a centralized architecture. Therefore, the global index used by each MR is extended from *AMMORE* index so that it contains information about where each operator is deployed in the network. Figure 6.2 shows an example of the global operator index.

Once MR is done with shared operators detection, it uses its mashup distribution planner to decide where (on which execution nodes) an incoming mashup would be executed, and if multiple nodes are involved in executing a mashup what part each would execute. The mashup distribution planning considers several factors including the mashup structure, the locations of the data sources and end-users, and the current performance of the overlay. The global performance monitor at MR interacts with the local performance monitors at individual

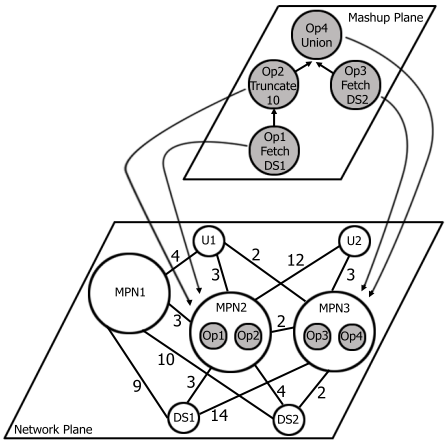

Figure 6.3: The operator placement problem in CoMap

MPNs, and it maintains a global snapshot of the overlay performance. The global synchronizers coordinate the activities of the MPNs involved in executing a mashup. Figure 6.1 also illustrates a mashup being executed on three MPNs.

Next, we explain our technique for distributing mashup execution and how it can adapt to network changes.

6.2 Planning Distributed Mashup Execution

This section formally describes the distributed mashup execution problem. After that, we explain how to plan the deployment of mashup operators in the overlay network and how they are executed in a distributed fashion.

6.2.1 Problem Statement

Figure 6.3 demonstrates the operator placement problem. We have a plane of mashups and a plane of network nodes, data sources, and end-users. System cost differs based on where operators are placed in the overlay network. Therefore, our goal is to place each operator in a node in the network such that system cost is minimum. Towards solving this problem, we

model the distributed mashup execution problem as an optimization problem. In *CoMaP*, we consider delay, bandwidth, and nodes load as metrics for computing the cost of executing operators in different places in the overlay network.

When a certain operator is executed on a given node, the cost of that execution is partitioned into computation cost and communication cost. Computation cost results from processing time of executing the operator on the node and communication time results from transmitting operators execution output to the next set of nodes that host the operators coming next in the mashup workflow. Suppose we have an operator Op_m that belongs to mashup Mp_n, created by end-user U_j, and hosted by MPN_k. The input of this operator is coming from the operators which are executed before Op_m, we refer to this set of operators as $PrvOpSet$ and we refer to some operator in this set as Op_q. We refer to the output size of Op_q as OS_q. The computation cost of Op_m deployed on MPN_k can be formulated as the summation of the size of the output data of each operator in $PrvOpSet$ in kilobytes divided by time needed by MPN_k to process each kilobyte of the data PT_k. Thus, $CompTime_{m,k} = \sum_{q=0}^{Q-1} \frac{OS_q}{PT_k}$.

Once Op_m operator finishes execution, it needs to send its output to the MPNs that host the next set of operators that are expecting input from Op_m. We refer to this set of MPNs as $NxtMPNSet$ and we refer to some MPN in this set as MPN_r. Communication time resulted by operator Op_m deployed on MPN_k can be formulated as the size of operator Op_m output in kilobytes OS_m divided by outgoing communication link bandwidth $BndLNK_{k,r}$ between MPN_k and each MPN_r in $NxtMPNSet$, the result is added to outgoing links communication delay per unit of data $DeLNK_{k,r}$ between MPN_k and each MPN_r. Therefore, $CommTime_{m,k} = \sum_{r=0}^{R-1}(\frac{OS_m}{BandLNK_{k,r}} + DeLNK_{k,r})$.

As a result, cost of execution of operator is the combination of computation and communication costs. Notice, that operators are executed multiple times according to their request rate, so, the total cost for an operator has to be multiplied with the operator's request rate RT_m. Therefore, cost of executing operator Op_m on MPN_k becomes $C_{m,k} = RT_m \times ((\sum_{q=0}^{Q-1} \frac{OS_q}{PT_k}) + \sum_{r=0}^{R-1}(\frac{OS_m}{BandLNK_{k,r}} + DeLNK_{k,r}))$.

System cost results from the combination of 1) Delay resulting from end-users $USet$ submitting their mashups to Mashup Controller MR 2) Delay resulting from MR performing operators merging on requested mashups 3) Delay resulting from MR assigning mashup operators to $MPNSet$ 4) Delay of executing mashup operators $OpSet$ on $MPNSet$. Cost in 4 can be optimized, therefore, our target in this optimization problem is to place operators $OpSet$ on $MPNSet$ such that total cost of executing operators is minimum. Let $Alloc_{m,k}$ be a {0,1} variable denoting that operator Op_m is deployed on MPN_k ($Alloc_{m,k} = 1$), otherwise, $Alloc_{m,k} = 0$. Therefore, the optimization problem is to assign values to each $Alloc_{m,k}$ variable such that $\sum_{m=0}^{M-1} \sum_{k=0}^{L-1} Alloc_{m,k} \times C_{m,k}$ is minimized while ensuring that the following constraints are not violated: 1) For an operator Op_m, $\sum_{k=0}^{L-1} Alloc_{m,k} = 1$, this indicates that Op_m is deployed only on one MPN; 2) $LD_k <= LdLimit_k$ which indicates that the load of MPN_k should not exceed its maximum allowed load.

A few naive approaches can be used to deploy mashup operators in overlay network, the first one is '*Random*' deployment where mashup operators are distributed randomly on MPNs. Another approach is '*Destination*' deployment where mashup operators are deployed on MPNs that are closest to the end-user who requested the mashup. The opposite approach is '*Source*' deployment where mashup operators are deployed on MPNs that are closest to data sources that contribute to these operators. One more approach is the '*Optimal*' deployment in which an exhaustive search is performed to deploy operators on MPNs that yield the minimum cost. The '*Optimal*' approach is expected to produce the best result but its running time is exponential due to its exhaustive search nature.

6.2.2 Our Scheme - DIMA

This subsection describes our approach for deploying mashup operators. Our approach is a two-stage optimization process where initial operator deployment is performed in the first stage and a migration process takes place in the second stage. We introduce a running example represented in Figure 6.4 throughout our discussion. In this example, a mashup

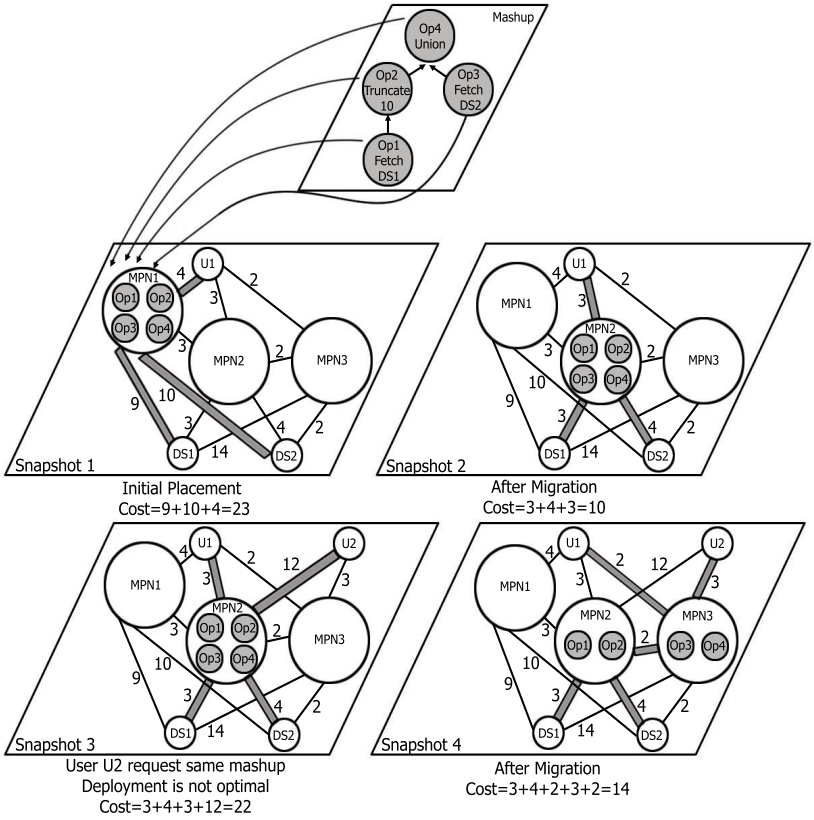

Figure 6.4: A scenario of CoMaP operator placement

consisting of four operators is to be deployed on MPNs. The mashup first uses a fetch operator (Op_1) to pull data from data source DS_1, then it uses Op_2 to truncate 10 items from the result of fetching data. Second, the mashup uses a fetch operator (Op_3) to fetch data from data source DS_2. Finally, the results of Op_2 and Op_3 are combined using a union operator (Op_4), and the final result is dispatched to end-user (U_1). Figure 6.4 consists of 4 parts representing in-order snapshots of the system. Link delays on the figure represent the final delay (in seconds) resulting from taking propagation delay, bandwidth, and data source sizes into consideration. For the sake of clarity in this figure, we assume all MPNs have the same processing power. One thing to mention is that geographical location of a node is reflected by its location in the layout. For example, by looking at part 1 of the figure, we

can see that MPN_1 is closer to U_1 than MPN_2. We can also see that MPN_2 is closer to U_1 than MPN_3 Communication links that contribute to system cost are shaded in grey.

Stage1: Initial Deployment

Mashup Controller MR has information about all MPNs in the system, including load of MPNs and their coordinates in the network distance graph. Such information is transferred to MR by MPNs upon joining the network. The load information for MPNs is then updated as MR deploys mashup operators on MPNs. Once an end-user sends his mashup to MR, the coordinates for the end-user machine is also sent to MR.

In this stage, when the mashup arrives to MR, MR uses end-user coordinates and MPNs coordinates to compute Euclidean distance between each pair of end-user and MPN. This distance is then used to estimate delay of sending data from MPN to end-user. Each operator is then initially deployed on the MPN that has the minimum delay from the end-user. In our running example, part 1 of Figure 6.4 shows the initial deployment stage for the four operators. Notice that delay in this stage is only computed based on coordinates (Euclidean distance). Here, since MPN_1 is geographically closer to U_1 than MPN_2 and MPN_3, all operators are initially deployed on MPN_1.

This initial operator deployment may not always be optimal for the following reasons. First, this stage depends on Euclidean distances to estimate delays which is not accurate as relying on current network conditions. In part 1 of our running example, actual link delays do not comply with geographical location of nodes. This can happen because some nodes reside on fast links, others might reside on slower links. It can also happen because of congested links. Second, operator deployment in this stage is based on distances from end-users, if distances from end-users and data sources are taken into consideration; better deployment decisions might be achieved. In part 1 of our running example, delays between MPN_1 and data sources DS_1 and DS_2 are not considered. However, distances from data sources are unknown at this stage because the coordinates of data sources are not known. The number of data sources on

the Web is huge, this is why the system cannot store and maintain coordinates of that large number of data sources. Third, as we will explain in the next subsection, mashup operator deployment becomes sub-optimal when more end-users share mashups.

The complexity of the initial deployment stage can be analyzed as follows, each operator is considered for positioning on each MPN, so the complexity is the number of MPNs (L) multiplied by number of operators (M). So, the complexity is $O(L * M)$. Once the initial placement of each operator is decided, the entry for that operator in the global mashup index is updated with the new deployment information that specifies at which MPN the operator is deployed. Although this initial step may not lead to optimal deployment decision, it is a good initial step. Next, the optimization process seeks to improve deployment by going through a migration process.

Stage2: Operator Migration

To minimize system cost, operators should migrate from the MPN on which they are deployed to a new MPN that leads to a better deployment decision. This migration process has to be fully distributed to preserve the scalability feature of *CoMaP*.

Each MPN has to decide if migrating operators to one of its neighbors decreases the total system cost; the total system cost is not available for MPN nodes due to the distributed nature of *CoMaP*. Basically, the total system cost is the summation of operator's execution costs all over the network. So, if each MPN minimizes operator execution cost within its neighborhood, that will eventually decrease the total system cost.

To compute the amount of cost change resulting from migrating operators from MPN_i to MPN_j, first, we introduce the cost of a migration *state*. Each migration step has two states; $CurrentSate$ and $NewState$ where the $CurrentState$ represents hosting operators on MPN_i (Before migration) and the $NewState$ represents hosting operators on MPN_i in addition to hosting migrated operators on MPN_j (After migration). Therefore, we have two costs; $CurrentStateCost$ and $NewStateCost$ where the cost of each state includes the

cost of hosting operators in that state; such that the cost of hosting an operator on some MPN is given as follows, 1) the cost of sending input to MPN by the nodes that host the children of this operator. 2) The cost of processing the input on MPN. 3) The cost of sending output by MPN to nodes that host parents of this operator. 4) The cost of communication between MPN and the end-users sharing the operator which is evaluated by pinging end-users machines. When MPN_i is considering migrating operators to MPN_j, it computes $NetB = CurrentStateCost - NewStateCost$ and it also considers all direct neighbors MPN_j (number of hops=1). After that, operators migrate to the neighbor with maximum positive $NetB$ value. A positive $NetB$ value indicates that this migration step leads to minimization of system cost.

Now, we demonstrate this stage in part 2 of our running example, MPN_1 is considering migrating all 4 operators to MPN_2. In this case, MPN_1 uses ping pong messages to estimate cost of fetching data from DS_1 and DS_2. Then, it uses ping pong messages to estimate cost of sending the result to U_1. Accordingly, $CurrentStateCost = 23$. Now, MPN_1 asks its neighbor MPN_2 about the cost of hosting the 4 operators on it. Here, MPN_2 uses ping pong messages to estimate the cost of fetching data from DS_1 and DS_2 plus the cost of sending result to U_1; then MPN_2 sends the result back to MPN_1. Based on MPN_2 feedback, MPN_1 finds out that $NewStateCost = 10$. After that, MPN_1 calculates the net benefit ($NetB = 13$) and decides to migrate all 4 operators to MPN_2.

Notice that if no such **direct** neighbor with $NetB > 0$ is found, MPN_i widens its search process by looking at indirect neighbors where the number of hops = 2. This increase in the tested neighborhood helps '$DIMA$' scheme to avoid local optima. At the same time, the maximum number of hops we use for the local search process is 3, it is kept low so that *CoMaP* efficiency is not degraded. Also, this parameter can be set by the system administrator.

Part 3 of our running example considers the case when a new user U_2 requests the same mashup which U_1 initially requested. Because U_1 and U_2 share mashups, the previous opera-

tors deployment which is based on U_1 requesting the mashup is not optimal any more. This happens because communication between MPN_2 and U_2 results in high delay. Therefore, migration is needed again. In this example, MPN_2 is considering migrating Op_3 and Op_4 to MPN_3 as a target neighbor. MPN_2 repeats the same migration steps which results in $NewStateCost = 14$, $CurrentStateCost = 22$, and $NetB = 8$. As a result, MPN_2 decides that Op_3 and Op_4 should migrate to MPN_3 which is reflected in part 4 of Figure 6.4.

When an operator migrates from MPN_i to MPN_j, MPN_i informs the MPNs on which children and parent operators are deployed with such a change. In addition, MPN_i informs the mashup controller about the new change. The mashup controller in turn updates its mashup index with the new deployment decisions. Therefore, when new requests for mashups arrive to the mashup controller, the controller is able to consult the up to date mashup index to find out to which MPNs the mashup execution should be directed.

The migration process is performed periodically to ensure that *CoMaP* adapts to newly requested mashups and to changes in the number of end-users sharing operators. Moreover, the migration process needs to be executed periodically to adapt to changes in network links delay and bandwidth. Note that communication costs between MPNs is calculated such that delay and bandwidth values are the actual values of the links in the overlay network.

The complexity of the migration process can be analyzed as follows, each migration happens within each node locality because each node considers migrating operators to one of their neighbors. Also, migration occurs on the subtree level, which means that subtrees of operators are what is in consideration of migration. So, we need to compute the number of subtrees in a mashup tree. The number of subtrees depends on two variables, namely, depth of mashup tree and number of branches in mashup tree. Since one fetch operator is the starting point of any branch in a mashup tree, the number of branches in a mashup tree equals to the number of fetch operators in that tree. Therefore, the number of subtrees equals number of branches (b) multiplied by tree depth (d). Now each subtree is considered for migration to one of the (h) neighbors. So, the total complexity is $O(b * d * h)$.

Notice that the cost of probing nodes done in the migration process is considered tolerable for the system because probing only occurs periodically when the migration process is performed. In addition, the migration process happens within each MPN locality which means that the migration process is performed smoothly without the system functionality being affected.

In the next section, we discuss failure resiliency which is an important quality for cooperative distributed information systems.

6.3 Failure Resiliency

The sources of failure in *CoMaP* could come from 1) Failure of Mashup Controller (MR) or 2) Failure of Mashup Processing Nodes (MPNs). Failure of a mashup controller is handled by replicating it which helps to avoid a single point of failure in the system. Among those controllers, one of them is the main controller and the other replicas are secondary controllers. All mashup controllers are identical to one another in terms of system information they posses and each one of them plays the same role in terms of receiving and handling end-user requests. However, the main controller plays slightly different role than secondary controllers in failure resiliency process. All mashup controllers are chosen to be distributed geographically in the network such that each controller serves the end-users closer to it.

To guarantee correct functionality of *CoMaP*, certain interaction between controllers is needed. For example, detecting shared operators requires that each controller knows about all operators in all mashups sent by end-users. Therefore, when one controller receives a mashup from an end-user, it directly sends the mashup information to all other controllers. This way, detecting shared operators becomes identical in all controller nodes. However, when a mashup is sent to a controller, that controller is the only one responsible of directing the execution of that mashup.

Another type of communication is needed between controllers in the case of controller failure.

Basically, each controller has one identical list of nodes which specifies three pieces of information. First, who is currently the main controller? Second, what is the current set of secondary controllers? Third, what is the set of candidate nodes that can be replacements of secondary controllers? The main controller exchange heart beat messages with secondary controllers to ensure they are still alive. If the main controller did not receive a reply from one of the secondary controllers, it assumes it failed and responds by performing the following operations. First, it uses the candidate set to assign a new secondary controller. Second, its current state is duplicated on to the new secondary controller so that it can start operating. Third, it notifies all other secondary controllers about the failure of the old controller and the existence of the new replacement.

Failure of the main controller is handled as follows, in case secondary controllers do not receive heart beat messages from the main controller, they assume it failed. Here, the current set of secondary controllers is used to recover from such a failure. Basically, what happens is that the current set of secondary controllers is ordered such that the first controller in the list has the responsibility of replacing the main controller when it fails. In this case, the previously mentioned secondary controller (new main controller) performs the following operations. First, it eliminates itself from the current secondary controllers list. Second, it propagates the change in this list to all other controllers. Third, it announces itself as the new main controller to all other secondary controllers. The introduction of coordinator replicas causes one change on '$DIMA$' operator deployment. When an operator migrates from MPN_i to another MPN, MPN_i contacts the coordinator in its area to inform it about operator migration. That coordinator in turn distributes the information to all other coordinators.

So far, we discussed failure within controllers, now, we discuss failure of Mashup Processing Nodes (MPNs). MPNs exchange heart beat messages with direct neighbors, if one MPN did not receive a reply from one of the neighbors, it assumes failure of that neighbor and reports the failure to the controller in its area. Once the controller receives the failure notification, it reallocates the operators which used to be hosted by the failed MPN to a new

MPN. This new allocation is propagated to the main controller and all secondary controllers. Moreover, if failure occurred to one of the mashup processing nodes (MPNs), then the effect of this failure is alleviated by replicating operators. If one MPN hosting an operator fails, then the operator can still be accessed through its replica. Since a huge number of operators exist in *CoMaP*, we cannot replicate each one of them. So, we select a percentage of the most overloaded MPNs in the system and we replicate their operators. This percentage is selected by the system administrator based on observed system performance. When one MPN replicates an operator to another MPN, it also informs the controller in its area of the replication process, and that controller propagates this change to all other controllers.

After proposing the technical contributions of this dissertation, we continue in the next chapter with a comprehensive experiments that show the benefits and overhead of using our architectures and protocols.

Chapter 7

Experiments and Results

The goals of our experiments are as follows, (1) evaluating the impact of common component detection on mashup execution; (2) evaluating the effect of operator reordering on the performance of mashup execution; (3) studying the impact of MACE's dynamic cache point selection on the performance of the mashup platform; 4) evaluating the benefits and overheads of the proposed cache point indexing scheme; 5) testing the impact of range queries and taxonomy awareness support on our system; 6) evaluating '$DIMA$' approach by showing its effect on the performance of *CoMaP* distributed mashup processing platform; 7) discussing the effect of applying failure resiliency on CoMaP. Goals 1 and 2 are demonstrated in Section 7.1. Goals 3, 4, and 5 are discussed in Section 7.2. Goals 6 and 7 are investigated in Section 7.3.

7.1 Mashup Performance Evaluation

The goals of this section's experiments are two fold; (1) evaluating the impact of common component detection on mashup execution; (2) evaluating the effect of operator reordering on the performance of mashup execution. First, we describe our prototype, and then we describe our experimental setup. After that, we analyze common component detection and operator reordering schemes.

7.1.1 Prototype Development

In our *AMMORE* prototype implementation, we used a package called ROME [101] as the base of implementing our own feed processing operators. ROME is a package designed specifically to handle basic manipulation of Atom and RSS feeds. In our prototype, mashups are described in sets and enter the system as XML files. These XML files contain all mashups, their operators, and their attributes. Data sources information such as URL and popularity are embedded within fetch operators. We have not designed a visual interface for our platform, so end-users send their mashup files to our mashup server who is responsible of parsing the file, executing its mashups, and sending results to end-users.

7.1.2 Experimental Setup

AMMORE implementation contains one server which hosts a mashup platform, this mashup platform hosts mashups that fetch data from several data sources distributed across the Web. In our experiments, the most popular 1500 data sources in Syndic8 [102] were used for building mashups. Syndic8 is a repository for RSS and Atom feeds. The mean value for latency to extract data from data sources is 0.6 seconds and the average number of items in these sources is 21 items. The average number of fetch operators per mashup is 2 and the average number of operators per mashup varies from 5 to 20 operators. The number of subscriptions to a data source is representative of its popularity which is also extracted from Syndic8. Data processing operators are distributed randomly on mashups. We perform our experiments with several mashup sets that consist of 2000-10000 mashups. In the following experiments, regular execution corresponds to executing every single component of the input mashup set. Also, the term delay refers to the delay of executing the mashup set in milliseconds.

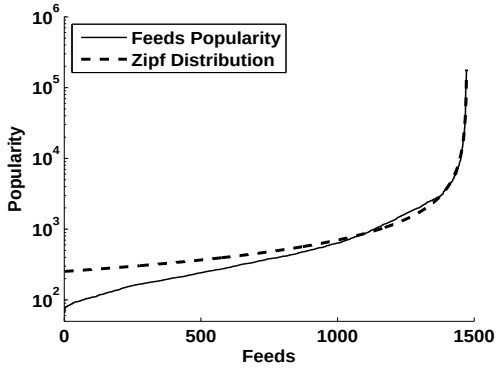

Figure 7.1: feeds popularity compared to Zipf distribution

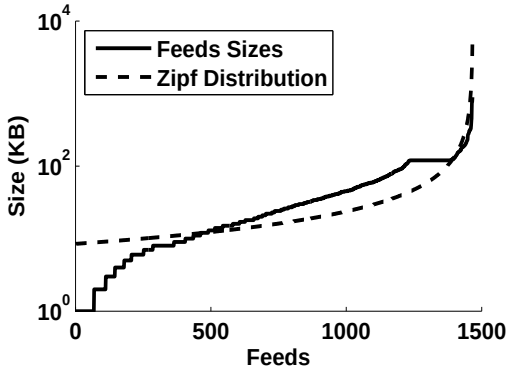

Figure 7.2: feed size compared to Zipf distribution

7.1.3 Feeds Characteristics and Trends

In this subsection, we state trends in the distribution of feeds over the Web. Finding these trends provides realistic information that can be used by researchers interested in Web 2.0. Feeds in this experiment are the most 1500 popular feeds on Syndic8. Figure 7.1 represents feed popularity as measured by the number of subscriptions it has on Syndic8. Popularity is compared to a Zipf distribution with α (exponent value) = 0.9. Results show that feeds popularity on the Web closely resembles the Zipf distribution. Similarly, Figure 7.2 indicates

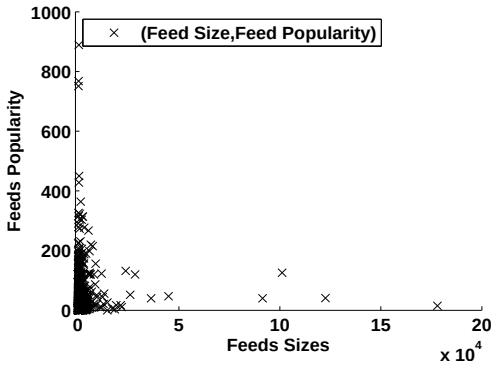

Figure 7.3: A plot of feeds data size and feeds popularity

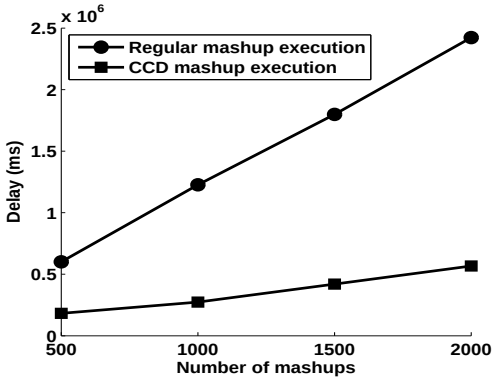

Figure 7.4: Delay of mashups execution when number of mashups is variable

that the feed sizes also resemble a Zipfian distribution with $\alpha = 0.9$. The relationship between feed popularity and feed size is represented in Figure 7.3 which shows that most feeds have small data sizes and low popularity.

7.1.4 Common Component Detection Results

Executing mashups without common component detection requires the mashup platform to execute every operator in every mashup which is time consuming. In Figure 7.4, 5 operators are used in each mashup and in Figure 7.5, 2000 mashups are used. For both experiments,

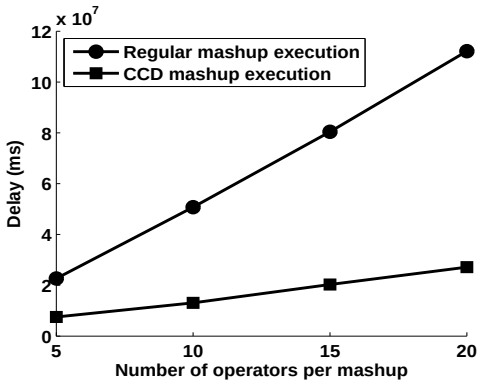

Figure 7.5: Delay of mashups execution when number of operators is variable

feeds are distributed on mashups based on a Zipf distribution with $\alpha = 0.9$. The standard deviations in Figure 7.4 are in the range [1.1, 1.5]. The standard deviations in Figure 7.5 are in the range [0.9, 1.5]. Both figures show that executing mashups with common component detection results in less delay because a subset of operators are detected as common across mashups.Accordingly, these common components are executed only once for each occurrence in all mashups. As a result, delay is minimized.

Figure 7.4 also shows that delay increases as the number of mashups increases. This is because more operators exist in the system which causes total mashup execution time to increase. The same behavior appears in Figure 7.5 because the number of operators is increasing which accordingly increases the time for executing mashups.

In Figure 7.6 we show the throughput of the system in terms of number of mashups executed per second. We can see that using common component detection increases the number of mashups executed per second. The standard deviations in Figure 7.6 are in the range [0.9, 1.5].

In Figure 7.7, the standard deviations are in the range [0.8, 1.3]. In this figure, we repeat the experiment in Figure 7.4, but we distribute feeds on mashups according to uniform distribution. The Figure shows that our technique is less effective in this case because selecting feeds

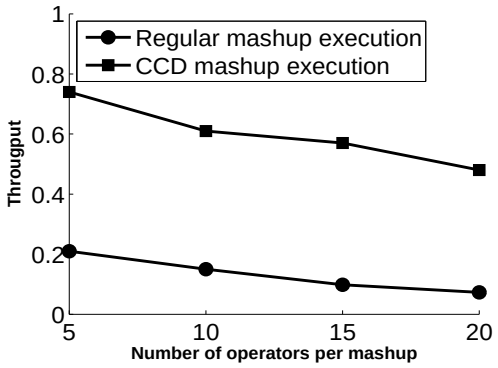

Figure 7.6: Throughput of the system in terms of number of mashups executed per second

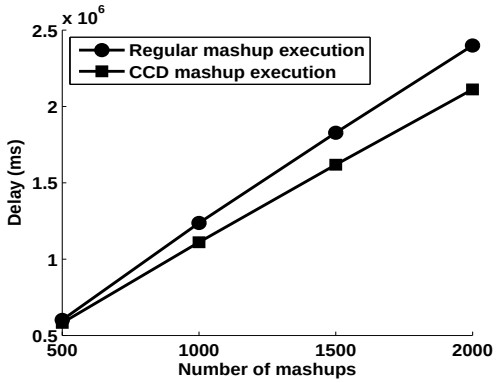

Figure 7.7: Delay of mashups execution when number of mashups is variable

randomly minimizes the probability of finding identical feeds across mashups which consequently hurts the common component detection process. However, as indicated by Figure 7.1, feed popularity on the Web resembles a Zipf distribution which increases the probability of our technique to find common feeds across mashups because few feeds have very high popularity and they are repeated extensively across mashups. The rest of experiments use a Zipf distribution for distributing feeds on mashups.

Merge percentage of common component detection (CCD) is plotted in Figure 7.8. We can notice that merge percentage increases as more mashups enter the system. The increase in

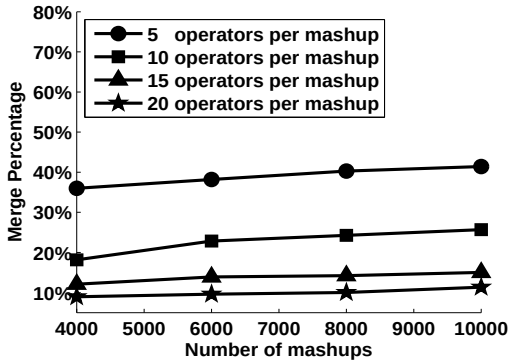

Figure 7.8: Merge percentage when using CCD

number of mashups creates a richer pool of components which can be a potential for detecting common components across mashups. On the other hand, the merge percentage decreases as number of operators per mashup increases, this happens because mashup depth increases. Recall that one of the conditions for two sequences of components to be considered common is that the operators in these sequences must appear in the same order. When mashups depth increases, depth of sequences of operators increases which accordingly decreases possibility of finding identical sequences of operators. In other words, detecting common sequences of operators consisting of only two operators is more likely to happen than detecting sequences of operators consisting of 8 operators.

In Figure 7.9 we notice that merging cost increases as the number of mashups increases which is a direct consequence of increasing number of operators in the system, but it is considered very small compared to delay savings. The standard deviations in Figure 7.9 are in the range [1.2, 1.4]. One point to mention here is that we expect the throughput of the system to increase with increasing concurrent end-user requests when CCD is used; this is because using common component detection decreases the total time needed for executing mashups.

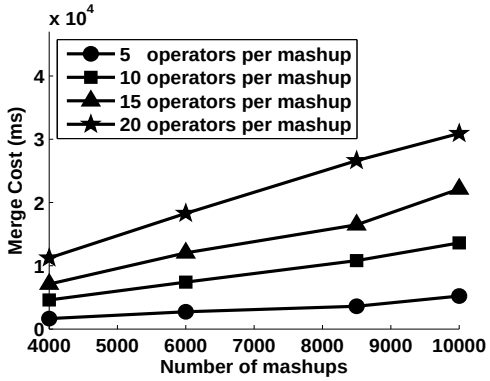

Figure 7.9: CCD Merge Cost when using different number of mashups and operators

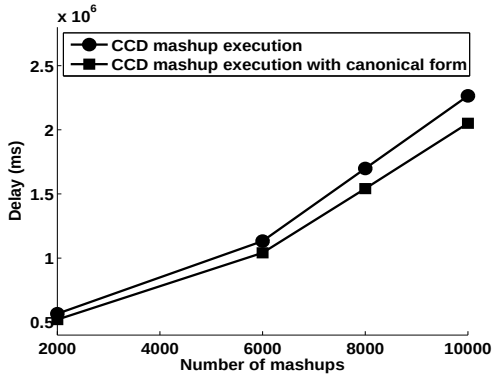

Figure 7.10: Delay of mashups execution when CCD is used with and without canonical form

7.1.5 Operator Reordering Results

The goal of having the canonical form is to transform mashups, so they are structured in a specific form. Doing this is expected to increase the effectiveness of common component detection. In Figure 7.10, 5 operators are used per mashup and the Figure shows that delay of mashup execution decreases when CCD with canonical form is used as opposed to the case when only CCD is used. This happens because as Figure 7.11 shows, merge percentage increases when canonical form is used along with CCD because having operators

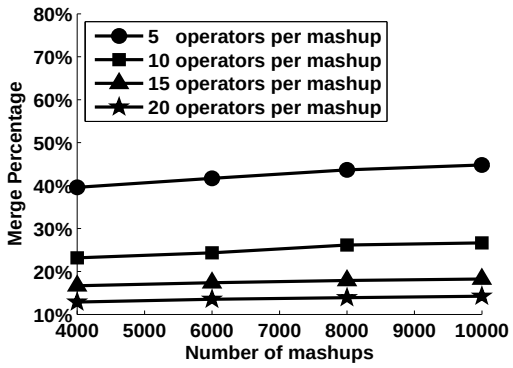

Figure 7.11: CCD Merge percentage when CCD and canonical form are used together

appear in a specific order in mashup trees implies that we have higher probability of having more operator sequences detected as common across mashups. One point to mention is that when the number of mashups increases, the merge percentage also increases because a richer pool of mashups exists as more mashups enter the system. When number of operators per mashup increases, the merge percentage decreases because mashup depth increases; when mashup depth increases, longer sequences of operators exist. The probability of detecting common operators in long operator sequences is lower than doing the same in shorter operator sequences. The standard deviations in Figure 7.10 are in the range [1.0, 1.4].

The experiment in Table 7.1 is performed on 2000 mashups and the standard deviations are in the range [0.9, 1.2]. One thing the table shows is the delay of executing our mashup set when our system applies common component detection (CCD) only. It also shows the same when our system applies common component detection (CCD) in addition to operators reordering (Canonical Form). We can notice that the delay in the later case is smaller than the delay in the earlier case. This behavior is an indication of the effect of operators reordering in standardizing mashups structure which increases the probability of detecting common components across mashups, and that in turn minimizes delay of executing mashups. Column 4 in the same table shows the delay of applying canonical form on our mashup set; this delay

Table 7.1: Delay (in milliseconds) of applying canonical form compared to its savings

Number of Operators	With CCD Only	With CCD plus Canonical	Canonical Delay
5	2,596,498	2,550,254	16
10	3,135,249	3,043,486	22
15	3,701,486	3,544,296	29
20	4,267,846	4,053,295	42

is very low compared to delay savings of executing mashups. The explanation of this is that canonical form is applied separately on each mashup which is a fairly low overhead task. Notice that as number of operators per mashup increases, the delay of executing mashups and applying canonical form increase as well. Having more operators in the system causes common component detection and canonical form to take more time to conclude.

7.2 Mashup Caching Evaluation

Our experimental study in this section has three objectives: 1) Study the impact of MACE's dynamic cache point selection on the performance of the mashup platform. 2) Evaluate the benefits and overheads of the proposed cache point indexing scheme. 3) Test the impact of range queries and taxonomy awareness support on our system. First, we describe the experimental setup.

7.2.1 Experimental Setup

Our experimental setup simulates a mashup environment with a mashup server, 80 data sources and 100000 end-users spread out on the Internet. The mashup server in our setup is, to a considerable extent, based upon the Yahoo Pipes environment [14]. Similar to Yahoo

pipes, our mashup platform contains 10 distinct operators namely, filter, sort, join, truncate, count, location-extraction, reverse, subelement, tail, and unique. End-users continuously create mashups which are executed on the mashup server. The mashup server executes the mashup and disseminates the results to the end-user.

We use two datasets for our experiments – a real dataset and a synthetic dataset. In the real dataset, we build our mashup to closely reflect reality. In this dataset, we have 5000 mashups which pull data from 80 real data sources over the Web. These data sources are extracted from syndic8 [102] feeds repository. When a mashup is executed, an actual connection to data sources is made to fetch data and the communication time needed to fetch data is measured. We also implemented a set of operators so that they process the real fetched data and therefore their execution time is also measured. In our real mashup set, the mean value for latency to extract data from data sources is 0.6 seconds and the average number of items in these sources is 21 items. The number of subscriptions to a data source is representative of its popularity. Recall from Figure 7.1 that feeds popularity closely resembles a Zipfian distribution with $\alpha = 0.9$.

Our synthetic dataset contains simulated operators where the execution time for each of these operators is estimated by performing a number of experiments on Yahoo pipes wherein we evaluated the latencies and output sizes of individual operators on XML feeds with sizes varying from 100 KB to 3 MB. As a result, we have realistic cost functions and output size functions. The number of distinct mashups existing at the platform in the synthetic dataset varies with the experiment, and it ranges from 1000 to 5000. The mean value for latency to extract data from data sources is 15 seconds and the average number of items in these sources is 536 items. For our synthetic dataset, the network topology is based upon the measurement by DIMES [103] on the actual Internet in 2008. We use BRITE [104] and BRITE extension [105] to transform DIMES data into a more convenient form. Our topology has 378444 nodes.

7.2.2 Evaluation of the Dynamic Cache Point Selection Scheme

In the first set of experiments, we quantify the performance benefits of the dynamic cache point selection scheme. The dynamic cache point selection scheme is compared to two other schemes: *End-results caching* wherein only the end-results of the mashups are cached, and *No caching* wherein the mashup platform does not employ any type of caching. These three schemes are compared with respect to the total cost incurred by the mashup platform in serving the end-user requests. For an individual mashup, the cost is quantified as the associated computational latency at the mashup platform.

In the first experiment, we compare the three schemes as the mean of the request rates of all mashups varies from 20 requests per unit time to 100 requests per unit time. The total number of mashups at the server is 5000 (therefore, the cumulative request rate at the mashup server varies from $10,000$ and $500,000$). In the synthetic dataset, a Zipfian distribution with $\alpha = 0.9$ is used to model the popularity variations among the individual mashups, while in the real dataset, popularity variations among individual mashups is extracted from syndic8 feeds repository. The mean of the update frequencies of the data sources (henceforth referred to as update frequency) is set to 60. This experiment is conducted on the synthetic dataset. The cache is assumed to have enough storage to hold the results (intermediate or final) of all mashups. Thus, we use the dynamic cache point selection algorithm for the no-storage limitations scenario. Figure 7.12 shows the total costs per unit time for the results of the experiments. As the results indicate, the cost incurred by the MACE's dynamic cache point is lower than the other two schemes throughout the simulated request rate range. The cost incurred by the End-results caching scheme is essentially constant as requests are served using cached data not requiring additional computations. In End-results caching, costs are mainly due to re-calculation of the cached results when one or more inputs used in a mashup changes[1]. At very low request rates, the costs of no-caching scenario are comparable to those

[1]First-time mashup executions also contribute towards the total costs in End-results caching but these costs are comparatively very small.

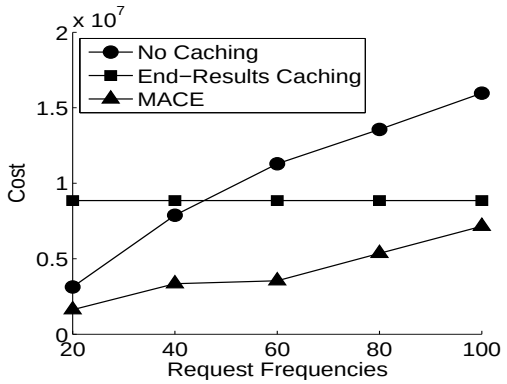

Figure 7.12: Total cost when request frequency is variable

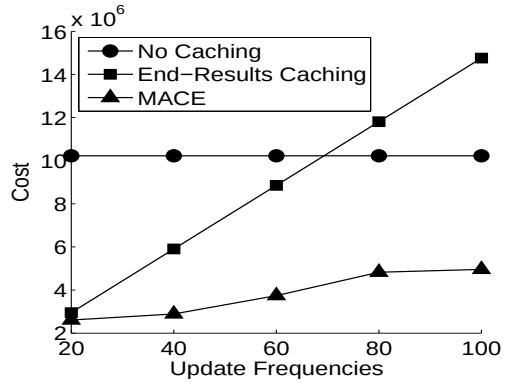

Figure 7.13: Total cost when update frequency is variable

of the MACE system. However, the costs of no-caching scenario rises quickly with increasing request rates. It is to be noted here that although the costs of the dynamic cache point selection scheme increases with increasing request rates, it does not rise indefinitely; its curve becomes flat once upon reaching the End-results caching cost levels.

In the second experiment (Figure 7.13) which is also conducted on the synthetic dataset, we study effect of update frequencies of data sources on the performances of the three schemes. The setup is very similar to that of the previous one except that the mean mashup request rate is fixed at 60 requests per unit time whereas the update frequency of all data sources is

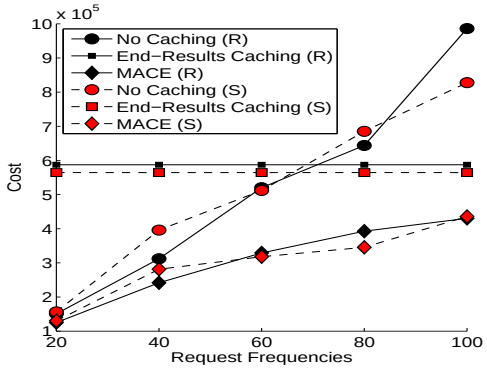

Figure 7.14: Total cost in the cases of synthetic and real data when request frequency is variable

varied from 20 to 100 per unit time. Again, we see that the MACE system yields significantly better performance than the other two schemes. However, in this experiment, the costs of the no-caching scenario remain constant. This is because, there is no cached data that needs to be recomputed when the input data changes.

In the third experiment, we aim to compare the results of experiments of the synthetic dataset with the results obtained from the real dataset. So, we measure the total cost per unit time required for executing our mashup set in both cases. First, we fix update rate to 60 updates per unit time and we vary request rate in the range of 20 to 100 requests per unit time. After that we fix request rate to 60 requests per unit time and we vary update rate in the range of 20 to 100 updates per unit time. In Figures 7.14 and 7.15, "R" refers to real dataset and "S" refers to synthetic dataset, these two figures show that the patterns we have for the real dataset are similar to the patterns we have for the synthetic dataset, the only thing different is the scale of cost values. The cost values for real data set in Figures 7.14 and 7.15 reflect the real world and make our experiments more realistic.

The better performance of the dynamic cache point selection scheme is essentially due to its ability to adapt to the changing update and request frequencies by moving the cache point to upper or lower levels of the tree. Figure 7.16 demonstrates this phenomenon by plotting

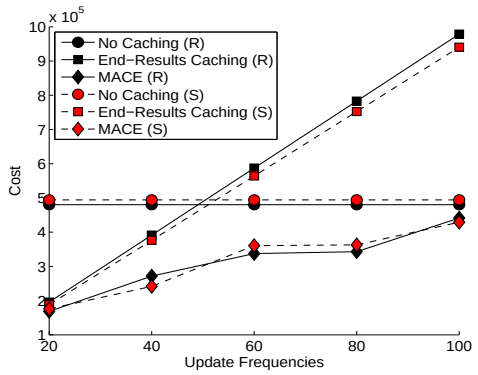

Figure 7.15: Total cost in the cases of synthetic and real data when update frequency is variable

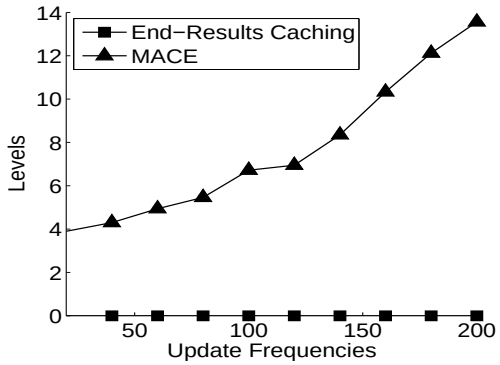

Figure 7.16: Level of Cache Points when update frequency is variable

the average level of the cache points as the update frequency varies from 20 to 100. The mean mashup request rate remains constant at 60. As the results indicate, as the update rate increases, MACE selects cache points that are located at lower-levels of the tree thereby reducing the costs of recomputing the cached results. The End-results caching, on the other hand, always caches at the same level (just before the dispatch operator).

In the next experiment, we use our synthetic dataset to evaluate the three scenarios when the storage available at the caches is limited. In this experiment, we fix the total request

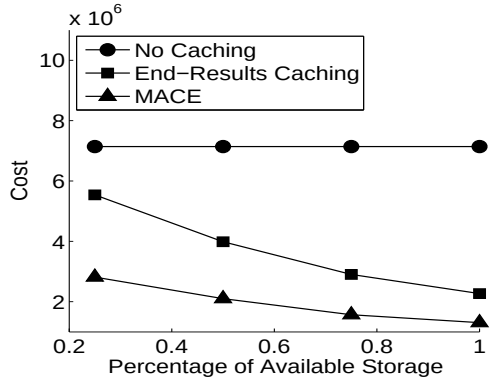

Figure 7.17: Total cost when available storage is variable

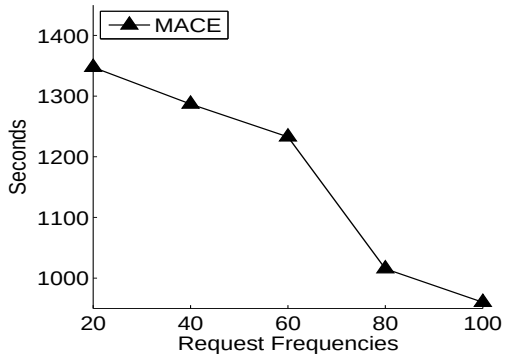

Figure 7.18: Total index access time when request frequency is variable

rate at 60 and update frequency at 180. The storage availability is varied from 10% to 100% of the storage needed for caching entire result set for the particular caching strategy. Least Recently Used cache replacement is employed for all schemes. As Figure 7.17 demonstrates, MACE results in better performance by selecting cache points that provide higher per-byte cost savings.

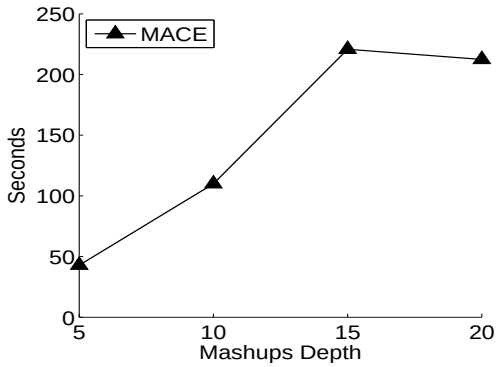

Figure 7.19: Total index access time when mashup depth is variable

7.2.3 Cache Index Analysis

In the second set of experiments, we use our synthetic dataset to study the scalability and performance of MACE's indexing mechanism by measuring the average latency involved in accessing a cache point stored in the B+ tree index. In the first experiment in this set, we evaluate the effects of request rate on the index access time. The mashup server has 5000 mashups with each mashup having 11 operators. The update frequency of all data sources is held constant at 60. As Figure 7.18 shows, index access time decreases as request frequency increases. This is due to two factors. First, when request frequency increases, MACE tends to select cache points near the roots of the respective mashup trees. As we move closer to the root, the width of the tree shrinks and the number of cache points in the index decreases. Second, MACE analyzes a new mashup starting from its root and goes down the tree looking for matching cache points. At high request frequencies, the cache points are closer to the root, and hence the search for matching cache points concludes faster. This result shows an important strength of our indexing scheme - it responds faster when the request rates are higher thereby improving the mashup platform's scalability.

Next, we study the effect of the mashup depth on index access time. The server again contains 5000 mashups. The mean mashup request rate and the update frequency are both

set to 60. Figure 7.19 shows the index access times when the depth of the mashups is varied from 5 to 20. Initially, the index access time increases linearly with mashup depth. The reason for this behavior is that probability of selecting cache points from lower levels of the tree increases as the mashup depth increases, and hence the search for matching cache points takes more time to conclude. However, the index access time becomes flat when the mashup depth reaches around 15.

7.2.4 Evaluation of Range Queries and Taxonomy awareness

In this experiment which is conducted on the real dataset, we study the effect of using partial matching on cache index utilization. The mashup server has 5000 mashups with each mashup having 11 operators. The update frequency of all data sources is held constant at 60. Figure 7.20 shows that using support for range queries decreases the total cost for executing mashups by 9 percent. This effect happens because cache index hit rate increases as Figure 7.21 shows. In the case where range queries are not supported, an index search might result in a cache miss. On the other hand, when range queries are supported, the probability of having a cache hit from searching the index increases. For the experiment of range queries, wherever we have a filter operator that filters data based on a numeric attribute, we use a numeric value between 1000 and 5000 upon which data is filtered (e.g; price < 3000).

The numeric value is selected randomly from the range $1000 - 5000$.

The effect of taxonomy awareness is shown in Figure 7.22 where we notice that the total cost for executing mashups decreases when taxonomy awareness is used. This also occurs because of the increase in cache index hit rate which is shown in Figure 7.23. An index lookup might result in a cache miss when taxonomy awareness is not used, but when it is used, the probability of having cache hits increases. We build our taxonomy by extracting keywords from Google search-based keyword tool [106]. This tool classifies the keywords people search for and put it in categories. The tool enables its users to search for keywords as well as

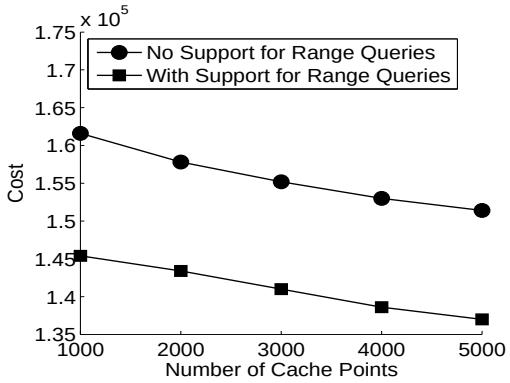

Figure 7.20: The effect of range queries support on the total cost

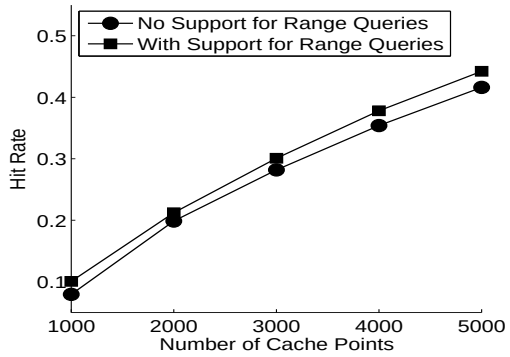

Figure 7.21: The effect of range queries support on index hit rate

download keywords categories and keyword lists as CSV files. When we build our mashups, wherever a filter operator is used, a random keyword or category name from the taxonomy is used as the parameter upon which data is filtered.

We believe that the percentage of improvement resulted from using the feature of range queries and taxonomy awareness is affected by the following. First, such a feature can only be applied to a subset of the operators that can be used to build mashups.

For example, range queries support can be used for filter, truncate, and tail operators, but it cannot be used for fetch, reverse, and unique operators. Second, the number of keywords

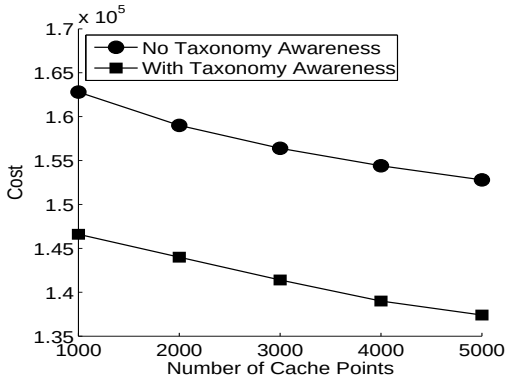

Figure 7.22: The effect of taxonomy awareness on the total cost

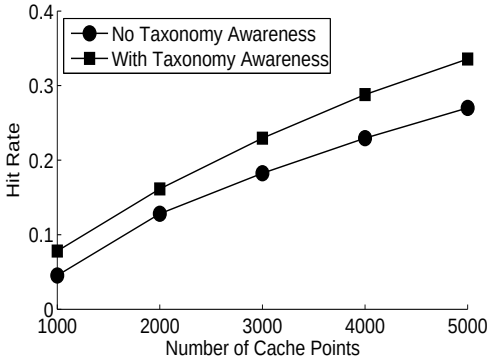

Figure 7.23: The effect of taxonomy awareness on index hit rate

end-users can choose from and the number of numeric values the end-users can use in their operators is high; therefore, the possibility of detecting operators where range query and taxonomy support can be used is low. Further improvement can be reached by considering the patterns end-users follow for selecting keywords and selecting numeric values in different domains. For example, if more end-users are interested in car prices below 5000, this can be an indication that most of end-user mashups related to filtering car results might contain price < 5000. The same principle is applied to taxonomy awareness, if end-users care most about

food, then more mashups are expected to use keywords related to food domain. Incorporating end-user patterns in building mashups is expected to produce more realistic results.

7.3 Distributed Mashup Execution Evaluation

We use simulation to perform our experiments. The goal of experiments is to evaluate '*DIMA*' approach by showing its effect on *CoMaP* performance. Also, we discuss the effect of applying failure resiliency on our system.

7.3.1 Experimental Setup

CoMaP environment simulates several operators which is similar to Yahoo pipes, these operators are Fetch, Filter, Sort, Union, Truncate, Tail, Sub-Element, Reverse, Unique, and Count. Our system consists of 4 mashup controllers, 100 data sources, 100 end-users, and a number of *MPN*s varying from 1000 to 4000. The total number of mashups requested by end-users varies from 1000 to 10000 where mashups request rate varies from 5 to 65 requests per unit time. The data sources are extracted from syndic8 [102] which is a repository for RSS and Atom feeds, the popularity distribution of data sources is also extracted from syndic8 where the number of subscriptions for a data source reflects its popularity. The number of operators per mashup varies from 10 to 40 where two of these operators are fetch operators and their data sources are selected based on data sources popularity, the rest of operators are selected randomly. Data source sizes vary from 1000 KB to 10000 KB. Our overlay topology is the Internet topology in 2008 measured by DIMES [103]. The number of nodes we use from this topology varies from 1204 to 4204.

7.3.2 System Evaluation

System cost in *CoMaP* is measured based on two factors, first, the average delay per mashup resulting from deploying and executing mashups operators, second, the average network usage per mashup which is defined as the average number of bytes transferred within the

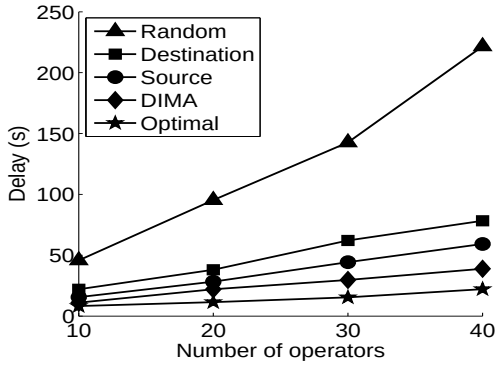

Figure 7.24: Average delay per mashup when number of operators varies

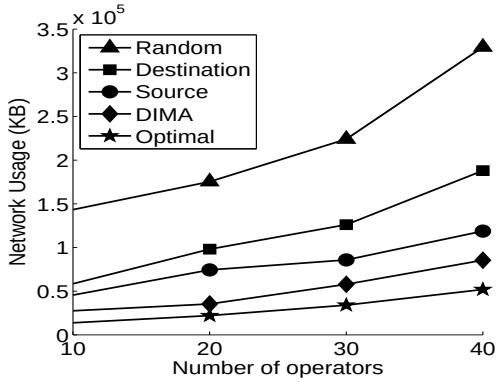

Figure 7.25: Average network usage per mashup when number of operators varies

network caused by executing mashups. The '$DIMA$' approach is compared to '*Random*', '*Source*', '*Destination*', and '*Optimal*' approaches. In all experiments we vary one parameter and keep the others constant. Unless mentioned, the constant values for number of mashups, number of operators, mashup request rate, data source size, and number of MPNs are 1000 mashup, 10 operators per mashup, 25 requests per unit time, 1000 KB, and 2000 MPNs, respectively.

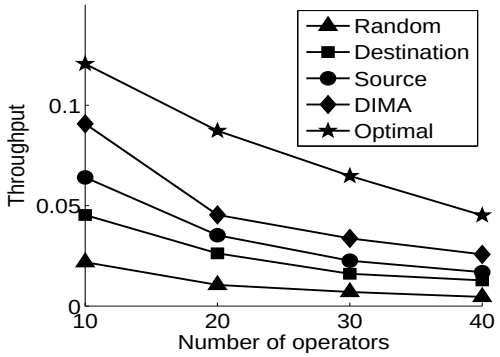

Figure 7.26: The throughput of the system in terms of number of mashups executed per second

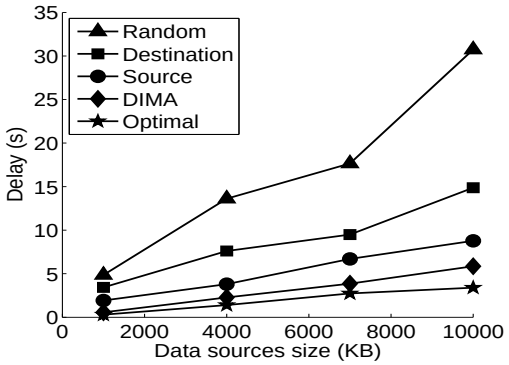

Figure 7.27: Average delay per mashup when data source size varies

In the first experiment we vary number of operators in *CoMaP* from 10 to 40 and we measure delay and network usage for the different schemes; Figures 7.24 and 7.25 show that '*Random*' scheme leads to high delays and high network usage and it is the worst among all schemes because it does not follow any kind of heuristics to deploy operators. The '*Source*' and '*Destination*' schemes lead to lower delay and network usage than the '*Random*' deployment. The results of the '*Source*' and '*Destination*' schemes might vary depending on how many end-users share operators. The '*Optimal*' scheme generates the lowest delay

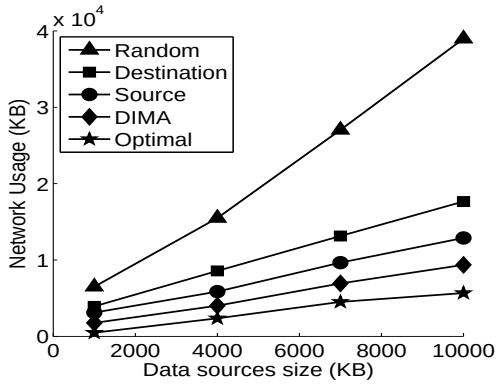

Figure 7.28: Average network usage per mashup when data source size varies

and network usage which is expected because of the exhaustive search performed by this scheme. This scheme is not practical because it requires long exhaustive search. '$DIMA$' approach beats '*Random*', '*Source*', and '*Destination*' approaches in terms of delay and network usage. This better performance is the result of a more dynamic two-stage optimization scheme that depends on distances from data sources and end-users at the same time, and it depends on operator migration which keeps *CoMaP* adapting to changes in network links delay and bandwidth and to changes in number of end-users sharing operators. Notice that '$DIMA$' performance is also close to the '*Optimal*' approach which proves its effectiveness. Figures 7.24 and 7.25 also show that the gap between each scheme and the '*Optimal*' scheme widens as more operators are used in mashups, this occurs because as more operators are used, more delay results normally from executing those operators. This delay is minimum in '$DIMA$'; because it uses migration to find better deployment options while migration is not used by the other schemes causing their delay of executing operators to increase rapidly. In Figure 7.26 we show the throughput of the system in terms of number of mashups executed per second. We can see that '$DIMA$' approach executes more mashups per second than the other approaches except the '*Optimal*' approach.

We performed another experiment where sizes of data sources varies from 1000 KB to

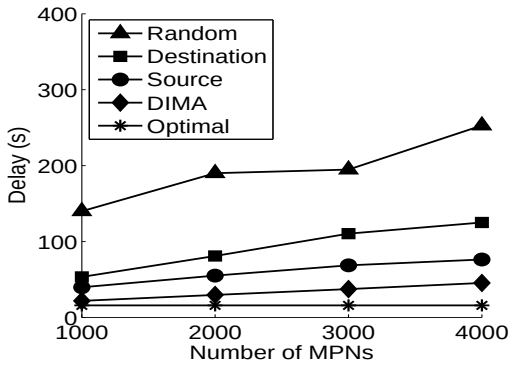

Figure 7.29: Average delay per mashup when number of MPNs varies

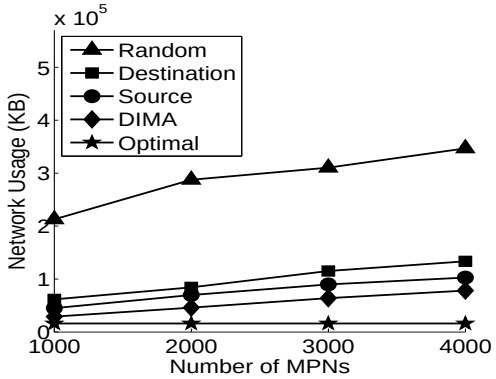

Figure 7.30: Average network usage per mashup when number of MPNs varies

10000 KB, as figures 7.27 and 7.28 show, the system cost increases for all schemes as data source sizes increase, this is due to increasing computation and communication costs resulted from increasing volume of data. In the next experiment, we vary number of MPNs in the network from 1000 to 4000 and measure delay and network usage. The results are plotted in Figures 7.29 and 7.30. The important point to take from these two figures is that the gap between '$DIMA$' scheme and '*Optimal*' scheme increases because as more MPNs are used, the search space size increases which adds more challenge for the '$DIMA$' scheme to find close to optimal deployment decisions.

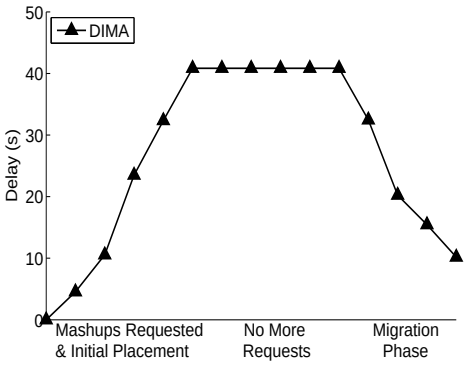

Figure 7.31: Average delay per mashup in different execution periods

We conducted an experiment to evaluate the effect of migration on system cost where we measure delay and network usage in three different periods of system execution. In the first period, mashups are requested, '$DIMA$' initial operator placement is executed (stage 1), and mashups are executed. The second period continues mashup execution without further requested mashups. The third period is when '$DIMA$' migration process (stage 2) starts and no further mashups are requested. As captured in Figures 7.31 and 7.32, delay and network usage increase when more mashups are requested in the first period, the system then stabilizes on the highest costs in the second period when no more mashups are requested, then delay and network usage drop significantly when the migration process starts. This cycle continues through the life time of *CoMaP*.

In the next experiment, we measure the overhead of enforcing failure resiliency in *CoMaP*. In Figure 7.33, failure probability is set to 20 percent, one mashup controller fails, number of mashups is set to $1,000$. The replication percentage varies between 10 percent and 70 percent which reflects the percentage of nodes their operators gets replicated. The figure shows the total overhead needed to maintain state of the system when applying failure resiliency. The overhead is measured in terms of network usage in kilobytes resulting from

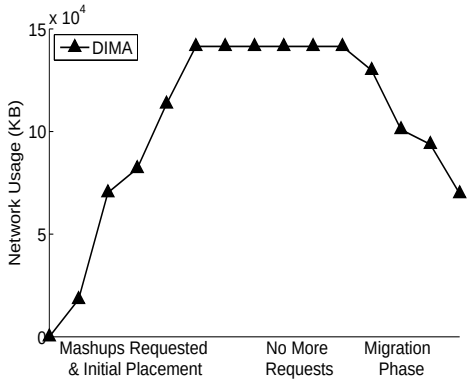

Figure 7.32: Average network usage per mashup in different execution periods

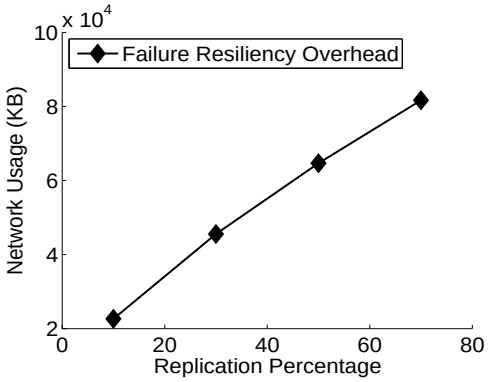

Figure 7.33: Total overhead of applying failure resiliency when replication percentage is variable

exchanged messages due to failure resiliency and replications. The state of the system includes keeping mashups information on mashup controllers identical. It also includes the cost of communication between controllers in case one of them fails. It also includes the communication between MPNs and controllers in case of replicating operators and failed MPNs. We notice that the overhead increases as replication percentage increases, this happens because more operators are replicated and therefore more communication occurs between MPNs and controllers. The system faces this kind of overhead only when the

system is initialized with replicas, during a failure, and when replicating operators. Other than these times, the system does not deal with this overhead.

The previous set of experiments show the effectiveness of the '$DIMA$' scheme in reaching operator deployment decisions leading to low network delay and usage. Despite of the overhead of applying failure resiliency in $CoMaP$, using it decreases the failure probability of $CoMaP$ by avoiding single point of failures.

Chapter 8

Conclusions

Mashups, while providing improved Web personalization, pose new scalability and performance challenges. Traditional Web service performance schemes are not effective for mashup domain due to its unique characteristics. In this dissertation, we proposed common component detection scheme which is used to reduce delay resulting from executing repeated mashup components. We also presented the use of a canonical form which increases the probability of detecting common mashup components and at the same time transforms mashups into an optimized form from efficiency stand point. In addition, we introduced a dynamic caching framework for mashups. This caching framework includes a dynamic mashup cache point selection scheme which maximizes the benefit of mashup caching. Finally, since most of the current mashup platforms are centralized, we designed a dynamic cooperative overlay-based mashup platform. This platform incorporates several novel features. First, we presented a scalable and efficient architecture comprising of a multitude of cooperative mashup processing nodes and a set of mashup controllers. Second, we introduced a dynamic mashup distribution technique that is sensitive to the relative locations of the sources and the destinations of a mashup, and optimizes data flow within the overlay. Third, we described how we enforce failure resiliency feature in our system. Load balancing is an important feature to be applied in our system as a future work. We conducted thorough and extensive experiments to study the benefits and costs of our architectures and schemes. Our experimental study demonstrated that our architectures and schemes yield improved system performance and scalability.

Bibliography

[1] Wikipedia, "Web 2.0," http://en.wikipedia.org/wiki/Web_2.0.

[2] Myspace Inc., "Myspace," http://www.myspace.com/.

[3] Facebook Inc., "Facebook," http://www.facebook.com/.

[4] Wikimedia Foundation Inc., "Wikipedia," http://www.wikipedia.org/.

[5] Google Inc., "Blogger," https://www.blogger.com/start, 1999.

[6] GNU, "Juice," http://juicereceiver.sourceforge.net/, 2005.

[7] Yahoo Inc., "Delicious," http://delicious.com/, 2003.

[8] Pandora Media Inc., "Pandora," http://www.pandora.com/, 2005.

[9] Youtube LLC., "Youtube," http://www.youtube.com/, 2005.

[10] Yahoo Inc., "Flickr," http://www.flickr.com/, 2005.

[11] ZOHO Corp., "ZOHO," http://www.zoho.com/, 2005.

[12] KAYAK, "KAYAK," http://www.kayak.com/, 2004.

[13] Google Inc., "Google reader," www.google.com/reader.

[14] Yahoo Inc., "Yahoo pipes," http://pipes.yahoo.com/, 2007.

[15] Intel Corp., "Mash maker," http://mashmaker.intel.com/web/, 2007.

[16] R. Hoegg, R. Martignoni, M. Meckel, and K. Stanoevska-Slabeva, "Overview of business models for web 2.0 communities," in *Proc. Workshop Gemeinschaften in Neuen Medien (GeNeMe)*, K. Meiner and M. Engelien, Eds. Dresden: TUDPress, Dresden, 2006, pp. 33–49.

[17] CNET Networks Inc., "Webware," http://www.webware.com/html/ww/100/2008/winners.html, 2008.

[18] A. Riabov, E. Bouillet, M. Feblowitz, Z. Liu, and A. Ranganathan, "Wishful search: interactive composition of data mashups," in *WWW*, 2008, pp. 775–784.

[19] IBM Corp., "Damia," http://services.alphaworks.ibm.com/damia/, 2007.

[20] N. Kulathuramaiyer, "Mashups: Emerging application development paradigm for a digital journal," *Journal of Universal Computer Science*, vol. 13, no. 4, pp. 531–542, April 2007.

[21] C. Jackson and H. J. Wang, "Subspace: secure cross-domain communication for web mashups," in *WWW*. New York, NY, USA: ACM Press, 2007, pp. 611–620.

[22] J. Wong and J. Hong, "Making mashups with marmite: towards end-user programming for the web," in *SIGCHI conference on Human factors in computing systems*, 2007, pp. 1435–1444.

[23] R. J. Ennals and M. N. Garofalakis, "Mashmaker: mashups for the masses," in *ACM SIGMOD international conference on Management of data*, 2007, pp. 1116–1118.

[24] X. Liu, Y. Hui, W. Sun, and H. Liang, "Towards service composition based on mashup," in *IEEE Congress on Services*, 2007, pp. 332–339.

[25] R. Tuchinda, P. Szekely, and C. Knoblock, "Building mashups by example," in *International Conference on Intelligent User Interfaces*, 2008, pp. 139–148.

[26] F. Dekeukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama, "Smash: secure component model for cross-domain mashups on unmodified browsers," in *WWW*, 2008, pp. 535–544.

[27] F. Majer, M. Nussbaumer, and P. Freudenstein, "Operational challenges and solutions for mashups- an experience report," in *Proceedings of second workshop on mashups, enterprise mashups, and lightweight composition on the web*, 2009.

[28] V. Hoyer and K. Stanoevska-Slabeva, "Design principles of enterprise mashups," in *Wissensmanagement*, 2009, pp. 242–253.

[29] A. Iyengar and J. Challenger, "Improving web server performance by caching dynamic data," in *USENIX Symposium on Internet Technologies and Systems*, 1997, pp. 49–60.

[30] J. Wang, "A survey of web caching schemes for the Internet," *ACM SIGCOMM Computer Communication Review archive*, vol. 29, no. 5, pp. 36 – 46, 1999.

[31] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglis, "Automatic Detection of Fragments in Dynamic Web Pages and its Impact on Caching," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 17, no. 6, pp. 859–874, 2005.

[32] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar, "Engineering web cache consistency," *ACM Transactions on Internet Technology (TOIT) archive*, vol. 2, no. 3, pp. 224 – 259, 2002.

[33] Q. Ly, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proceedings of the International conference on supercomputing*, July 2002, pp. 84–95.

[34] P. Wei and L. Xicheng, "AHBP: An efficient broadcast protocol for mobile Ad hoc networks," *Journal of computer science and technology*, vol. 16, no. 2, pp. 114–125, 2001.

[35] F. Luccio, A. M. Enriquez, P. O. Rieumont, and L. Pagli, "Exact rooted subtree matching in sublinear time," Universita Di Pisa, Tech. Rep., 2001.

[36] R. Cole, R. Hariharani, and P. Indyk, "Tree pattern matching and subset matching in deterministic o(n log3 n)-time," in *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, 1999, pp. 245–254.

[37] W. Liang and H. Yokota, "A path-sequence based discrimination for subtree matching in approximate xml joins," in *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops*. Washington, DC, USA: IEEE Computer Society, 2006, p. 116.

[38] A. Gupta, B. Liskov, and R. Rodrigues, "Efficient routing for peer-to-peer overlays," in *Conference on Symposium on Networked Systems Design and Implementation*. San Francisco, California, USA: USENIX Association, 2004, pp. 9–9.

[39] S. Ratnasamy, S. Shenker, and I. Stoica, "Routing algorithms for dhts: Some open questions," in *Peer-to-Peer Systems*. Springer Berlin / Heidelberg, 2002, pp. 45–52.

[40] P. Paul and S. V. Raghavan, "Survey of multicast routing algorithms and protocols," in *international conference on Computer communication*. Mumbai, Maharashtra, India: Springer Berlin / Heidelberg, 2002, pp. 902–926.

[41] C. D. Pham and C. Albrecht, "Optimizing message aggregation for parallel simulation on high performance clusters," in *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE Computer Society, 1999, pp. 76–83.

[42] D. Kranzlmller, P. Kacsuk, and J. Dongarra, "Recent advances in parallel virtual machine and message passing interface," in *International Journal of High Performance Computing Applications*, vol. 19, no. 2, 2005.

[43] D. Salomon, "Data compression: The complete reference." Springer, 2004.

[44] J. Zobel and A. Moffat, "Adding compression to a full-text retrieval system," *Software–Practice and Experience archive*, vol. 25, no. 8, pp. 891–903, 1995.

[45] M. Ouzzani and A. Bouguettaya, "Query Processing and Optimization on the Web," *Distributed and Parallel Databases archive*, vol. 15, no. 3, pp. 187–218, 2004.

[46] S. Pertet and P. Narasimhan, "Causes of Failure in Web Applications," Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep.

[47] G. Attiya and Y. Hamam, "Task allocation for maximizing reliability of distributed systems: a simulated annealing approach," *Journal of Parallel and Distributed Computing archive*, vol. 66, no. 10, pp. 1259–1266, 2006.

[48] K. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM archive*, vol. 36, no. 12, pp. 37–53, 1993.

[49] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," in *ACM SIGCOMM Computer Communication*, 2002, pp. 181–184.

[50] T. Loukopoulos and I. Ahmad, "Static and adaptive distributed data replication using genetic algorithms," *Journal of Parallel and Distributed Computing archive*, vol. 64, no. 5, pp. 1270–1285, 2004.

[51] L. Massouli, A. Kermarrec, and A. Ganesh, "Network awareness and failure resilience in self-organizing overlay networks," in *Geographic Load Balancing for Scalable Distributed Web Systems*, 2000, pp. 20–20.

[52] V. Cardellini, M. Colajanni, and P. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet Computing archive*, vol. 3, no. 3, pp. 1270–1285, 1999.

[53] C. Shahabi and Y.-S. Chen, "Web information personalization: Challenges and approaches," in *Databases in Networked Information Systems.* springer, 2004.

[54] B. Mobasher, R. Cooley, and J. Srivastava, "Automatic personalization based on Web usage mining," *Communications of the ACM archive*, vol. 43, no. 8, pp. 142 – 151, 2000.

[55] J. Srivastava, R. Cooley, M. Deshpande, and P. Tan, "Web usage mining: discovery and applications of usage patterns from Web data," *ACM SIGKDD Explorations Newsletter archive*, vol. 1, no. 2, pp. 12 – 23, 2000.

[56] M. Eirinaki and M. Vazirgiannis, "Web mining for web personalization," *ACM Transactions on Internet Technology (TOIT) archive*, vol. 3, no. 1, pp. 1 – 27, 2003.

[57] A. Datta, K. Dutta, D. VanderMeer, K. Ramamritham, and S. Navathe, "An architecture to support scalable online personalization on the Web," *International Journal on Very Large Data Bases (VLDB)*, vol. 10, no. 1, pp. 104 – 117, 2001.

[58] B. Mobasher, H. Dai, T. Luo, and M. Nakagawa, "Effective personalization based on association rule discovery from web usage data," in *Workshop On Web Information And Data Management archive*. ACM, 2001, pp. 9 – 15.

[59] C. Shahabi, F. Kashani, Y. Chen, and D. Mcleod, "Yoda: An accurate and scalable web-based recommendation system," in *International Conference on Cooperative Information Systems*, 2001.

[60] B. Mobasher, R. Cooley, and J. Srivastava, "Creating adaptive web sites through usage-based clustering of urls," in *Workshop on Knowledge and Data Engineering Exchange*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 19 – 19.

[61] N. Good, J. B. Schafer, J. A. Konstan, A. Borchers, B. Sarwar, J. Herlocker, and J. Riedl, "Combining collaborative filtering with personal agents for better recommendations," in *national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial*

intelligence. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1999, pp. 439 – 446.

[62] F. Liu, C. Yu, and W. Meng, "Personalized web search by mapping user queries to categories," in *Conference on Information and Knowledge Management archive*. New York, NY, USA: ACM, 2002, pp. 558 – 565.

[63] N. Henze and M. Kriesell, "Personalization functionality for the semantic web: Architectural outline and first sample implementations," in *International Workshop on Engineering the Adaptive Web (EAW)*, 2004.

[64] I. Cingil, A. Dogac, and A. Azgin, "A broader approach to personalization," *Communications of the ACM archive*, vol. 43, no. 8, pp. 136 – 141, 2000.

[65] D. Thaler and C. Ravishankar, "Using name-based mappings to increase hit rates," *ACM Transactions on Networking (TON) archive*, vol. 6, no. 1, pp. 1–14, 1998.

[66] Vinod Valloppillil and Keith W. Ross, "Cache array routing protocol v1.0.internet draft," 1997.

[67] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson, "Adaptive web caching: towards a new global caching architecture," *Computer Networks and ISDN Systems archive*, vol. 30, no. 22, pp. 2169 – 2177, 1998.

[68] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *ACM Transactions on Networking (TON) archive*, vol. 8, no. 3, pp. 281 – 293, 2000.

[69] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay, "Beyond hierarchies: Design considerations for distributed caching on the internet," in *International Conference on Distributed Computing Systems (ICDCS)*, 1999, pp. 273–273.

[70] A. G. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari, "Scalable Consistency Maintenance in Content Distribution Networks Using Cooperative Leases," *IEEE Transactions on Knowledge and Data Engineering archive*, vol. 15, no. 4, pp. 813–828, 2003.

[71] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar, "Improving Availability and Performance with Application-Specific Data Replication," *IEEE Transactions on Knowledge and Data Engineering archive*, vol. 17, no. 1, pp. 106 – 120, 2005.

[72] S. Shah, K. Ramamritham, and P. Shenoy, "Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers," *IEEE Transactions on Knowledge and Data Engineering archive*, vol. 16, no. 7, pp. 799 – 812, 2004.

[73] P. Rodriguez, C. Spanner, and E. W. Biersack, "Analysis of web caching architectures: hierarchical and distributed caching," *ACM Transactions on Networking (TON) archive*, vol. 9, no. 4, pp. 404 – 418, 2001.

[74] J. Baek, K. Gurpreet, and J. Yang, "A New Hybrid Architecture for Cooperative Web Cache," *Journal of Ubiquitous Convergence Technology*, vol. 2, no. 1, pp. 1 – 11, 2008.

[75] L. Y. Cao and M. T. zsu, "Evaluation of Strong Consistency Web Caching Techniques," *World Wide Web archive*, vol. 5, no. 2, pp. 95 – 123, 2002.

[76] B. Benatallah and H. R. Motahari-Nezhad, "Servicemosaic project: modeling, analysis and management of web services interactions," *Asia-Pacific conference on Conceptual modelling*, vol. 53, pp. 7–9, 2006.

[77] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Trans. on Software Engineering*, vol. 33, no. 6, pp. 369–384, June 2007.

[78] M. Aoyama, S. Weerawarana, H. Maruyama, C. Szyperski, K. Sullivan, and D. Lea, "Web services engineering: promises and challenges," *International Conference on Software Engineering*, pp. 647–648, 2002.

[79] J. Tatemura, O. Po, A. Sawires, D. Agrawal, and K. S. Candan, "Wrex: A scalable middleware architecture to enable xml caching for web-services," *Middleware*, pp. 124–143, 2005.

[80] D. B. Terry and V. Ramasubramanian, "Caching xml web services for mobility," *ACM Queue*, vol. 1, no. 3, pp. 70–78, May 2003.

[81] H. Artail and H. Al-Asadi, "A cooperative and adaptive system for caching web service responses in manets," in *International Conference on Web Services (ICWS)*. IEEE, 2006, pp. 339 – 346.

[82] C. Dorn and S. Dustdar, "Achieving web service continuity in ubiquitous mobile networks the srr-ws framework," in *International Workshop on Ubiquitous Mobile Information and collaboration Systems (UMICS)*, 2006.

[83] V. Ramasubramanian and D. B. Terry, "Caching of XML Web Services for Disconnected Operation," Microsoft Corp., Tech. Rep.

[84] X. Liu and R. Deters, "An efficient dual caching strategy for web service-enabled pdas," in *ACM Symposium on Applied Computing*, 2007, pp. 788–794.

[85] K. Devaram and D. Andresen, "Soap optimization via client side caching," in *ICWS*, 2003.

[86] T. Takase and M. Tatsubori, "Efficient web services response caching by selecting optimal data representation," in *International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2004, pp. 188 – 197.

[87] M. Mahdavi, B. Benatallah, and F. Rabhi, "Caching dynamic data for e-business applications," in *International Conference on Intelligent Information Systems (IIS)*, 2003.

[88] C. Doulkeridis, V. Zafeiris, K. Norvag, M. Vazirgiannis, and E. A. Giakoumakis, "Context-based caching and routing for P2P web service discovery," *Distributed and Parallel Databases archive*, vol. 21, no. 1, pp. 59 – 84, 2007.

[89] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *ICDE*, 2006.

[90] Y. Ahmad, J. Jannotti, E. Zgolinski, and S. Zdonik, "Network awareness in internet-scale stream processing," *IEEE Data Engineering Bulletin*, vol. 28, no. 1, pp. 63–69, 2005.

[91] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *CIDR*, 2005, pp. 277–289.

[92] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. etintemel, Y. Xing, and S. Zdonik, "Scalable distributed stream processing," in *CIDR*, 2003.

[93] S. Roy, H. Pucha, Z. Zhang, Y. C. Hu, and L. Qiu, "Overlay node placement: Analysis, algorithms and impact on applications," in *International Conference on Distributed Computing Systems (ICDCS)*, 2007, pp. 53–53.

[94] A. Cirillo, R. Jagadeesan, C. Pitcher, and J. Riely, "Tapido: Trust and authorization via provenance and integrity in distributed objects (extended abstract)," in *ESOP*, 2008, pp. 208–223.

[95] B. J. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," in *International Workshop on Information Processing in Sensor Networks (IPSN)*, 2003, pp. 47–62.

[96] O. Papaemmanouil, U. Çetintemel, and J. Jannotti, "Supporting generic cost models for wide-area stream processing," in *ICDE*, 2009, pp. 1084–1095.

[97] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *PODS*, 2005, pp. 250–258.

[98] S. Chandrasekaran, J. A. Miller, G. S. Silver, B. Arpinar, and A. P. Sheth, "Performance analysis and simulation of composite web services," *Electronic Markets: The International Journal (EM)*, vol. 13, no. 2, pp. 120–132, 2003.

[99] J. Wang, "A survey of web caching schemes for the internet," *ACM SIGCOMM*, vol. 29, no. 5, pp. 36–46, October 1999.

[100] E. T. S. Ng and H. Zhang, "A network positioning system for the internet," in *USENIX Annual Technical Conference*, 2004.

[101] Sun Microsystems Inc., CollabNet Inc., and Cognisync llc, "Rome," https://rome.dev.java.net/.

[102] Jeff Barr and Bill Kearney, "syndic8 feeds repository," http://www.syndic8.com.

[103] Y. Shavitt and E. Shir, "Dimes: let the internet measures itself," *ACM SIGCOMM*, vol. 35, no. 5, pp. 71 – 74, May 2005.

[104] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: an approach to universal topology generation," in *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, August 2001, pp. 346–353.

[105] M. Wahlisch, T. C. Schmidt, and W. Spat, "What is happening from behind? - making the impact of internet topology visible," *Campus-Wide Information Systems*, vol. 25, no. 5, pp. 392–406, 2008.

[106] Google Inc., "Search-based keyword tool," http://www.google.com/sktool/, 2008.