MANAGING CACHE FOR EFFICIENT QUERY PROCESSING

GOH SHEN TAT
(*MSc. (School of Computing)*)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2004

Acknowledgments

My thanks go out to my supervisor, Prof. Tan Kian Lee for his guidance as well as that special trust in my judgment and ability.

Also, I would like to thank my thesis advisers Prof. Sung Sam Yuan and Dr. Stephane Bressan for their valuable advices and comments.

To all my friends who have walked alongside with me during my most difficult days and have unreservedly shared all their joy with me, I give you my heartfelt thanks and my best wishes go out to you.

Lastly, to my mum and sisters who have stood by me all this while, and have given me all the encouragements and strength I needed to walk this far, I am eternally grateful.

Contents

List of Figures

List of Tables

Summary

This thesis discusses the opportunities and mechanisms to leverage query processing performance using caches. In a multi-user environment, it is common for users to have similar and repeated queries. Consequently, these queries can be satisfied more efficiently by introducing caches for keeping copies of answers nearer to the users. The profusion in greater storage spaces leads to more caches being made available at the clients and servers, and this has allowed the manifestation of algorithms to improve scalability, reliability and performance.

Caches can be managed in different ways, especially when the stored content can be accessed or manipulated at the hosts. In this thesis, we will begin by examining existing techniques that use caches to improve query processing. This will be followed by a discussion on some interesting research problems and a proposal on novel techniques to alleviate the problems.

Through our literature survey, we have identified a few interesting problems in managing and materializing data in caches for answering queries both in the centralized and

distributed environments. Efficient solutions to these problems will be introduced in the remaining of this summary and the details will be presented in the later chapters.

In our solutions, we have proposed methods to improve the mechanism for processing queries using caches. For each method, the design and performance have been thoroughly described and examined, each complete with detailed experimental studies and analysis. We have carried out in-depth exploration with these methods and have shown that they can contribute significantly in improving the caching mechanisms.

Our first research proposal was realized in a centralized multi-user environment where we have proposed a novel method using demand-driven caching. Such an approach is essentially non-speculative: the exact cost of investment and the return on investment are known, and the cache is certain to be reused. Three different algorithms were proposed and evaluated: Conform-CoD, Scramble-CoD and Integrated-CoD. We have also presented additional enhancements to extend the CoD mechanism. There, we examined the possibility of exploiting intra-query parallelism when large memories are available and the possibility of expanding the search space of CoD virtual caches for better overall performance.

Next, we moved on to a distributed environment, in a Peer-to-Peer system, and explored the caches for assisting query routing and answering. We have proposed two strategies. The first strategy promotes the reuse of the cache content and the second improves the query response time through recall-routing. These two strategies were realized through salvaging cache-content and caching visiting-queries. We have conducted

detailed experimental studies and evaluated these strategies in our work.

Lastly, we have proposed an efficient cache coherence method to update the caches for a hierarchy of network servers where each server serves a cluster of users. In this proposal, two mechanisms were proposed and evaluated: data scoping and delta profiling. These mechanisms work in conjunction with both the eager and lazy update models as defined generally in the literature.

With this thesis, we would like to report the findings that we have observed from the literature surveys, propose a list of efficient algorithms and mechanisms to improve the query processing performance using caches and present the results that we have achieved through the experiments. Finally, we hope that this thesis will be useful and relevant in some ways to researchers working in the area of managing cache for query processing.

Chapter 1

Introduction

1.1 Introduction

The profusion in storage spaces available at the clients and servers, no matter what the quantity or speed, has allowed the manifestation of algorithms to improve scalability, reliability and performance. In this thesis, we will first look at some of the existing techniques to improve query processing using caches. It will be followed by a discussion on some interesting research problems and a proposal on techniques to improve the performance. Through our literature survey, we have identified a list of problems in managing and materializing data in caches for answering queries both in the centralized and the distributed environments. Efficient solutions to these problems will be presented in details, in the main chapters.

1.2 Motivation

Query processing is ubiquitous in many multi-user applications. It aims to provide answers accurate and fast whenever possible. In many situations, a quick observation shows that many of these queries are repeated and hence require the same answer. Consequently, the queries can be satisfied more efficiently by introducing storages or caches for keeping copies of these answers nearer to the users. These storages can be caches or replicas. Their definitions can be found readily in the literature, with some crossing overs. Here, we briefly re-iterate some of the characteristics of these storages. Caches are mostly found on client machines which are populated by answers for queries. They are usually volatile and smaller in size compared to replicas. Replicas on the other hand, are created through the demand from cluster of machines and are usually server-oriented meant for serving a group of users. They are usually structured and require updating with the latest changes from the origin. In this work, we will mainly focus on how caches are managed and manipulated for improving the ways we answer queries for multiple users in different working environments.

The remaining part of this section aims to give a simple analogy to what our work is about before we move on to the details in the other Chapters. The analogy gives a complete picture of the motivation behind the proposed algorithms and what we aim to achieve with them. We illustrate our designs by following a series of situation extracted through the planning of a traveling itinerary.

Prior to any holiday trip, we would normally layout an itinerary or a list of places we will be visiting. On top of that, we often have side-notes on additional information of the places. The information comes from learned tips and directions through travel guide-books, friends, TV programs etc. Overall, a well-informed trip is desirable if it requires little effort to prepare or the additional effort is rewarded with a faster preparation.

In the first example, let's assume a character John who is going on a trip to a place that you have scheduled to visit upon his return, you could take advantage of this situation. First, you notify him that you will be going to the same place, and request that he keeps tag of all the information he comes across that might be useful to you. As an illustration, he could be looking for a cheap hotel in a particular town and might take him some time to look around and to ask the locals about good choices and locations. Needless to say, it will require him some effort to take note of the things that you have requested. But it is something that he would have to do for himself anyway. We assume that he will not be running any special errand for you. All these notes that he has taken down will save you a lot of time when you visit the same place, otherwise, you will have to do a lot of navigation on your visit. In our work, we have designed an algorithm to save these repeated effort similar to what has been illustrated through this example.

Next, on the assumption that you have not made concrete plans for a forthcoming holiday. You have decided that it will be valuable to collate information from people whom you know are making plans for their holiday trips. Meanwhile, you are approached by people who feel that you can contribute to their process of information collection. Fur-

ther, regardless of your contribution, you note down the popular destinations sought by them. After some time, you have finally decided on a place. So, with all the information at hand, you sieve for pieces that are useful to you. From your collection of information, some of the people that you have taken note of, might have gone for their vacations and returned. If indeed they had went to places which they have asked you before, then you have just found a source for questions that you might have for your trip. Otherwise, if they had decided not to go, then chances are that they still have information regarding people who have visited that place. This is so as they probably had spoken to many of these people while seeking for information, and this information is useful to you, hence giving you promising directions and probably saving valuable time.

Upon your return from your trip, you may now have collected more and the most updated information (travel maps, guidebooks and other information etc). However, you have decided to trash these items. Clearly, it is a pity to just throw these materials which are usually collected over a period of time. Beside, these materials could be useful to others who are likely to travel to these places. So, why trash these materials when you can give these items to people who will find them useful? In our work, we have designed an algorithm to salvage and recycle these materials, and have shown that these behaviors can be studied and translated to benefit users in a Peer-to-Peer environment.

Lastly, we direct our attention to the characteristics of an active telephone hot-line example for a cluster of travel groups. We assume that each group has a list of favorite destination checkpoints. Also, let’s assume Mary who is their assigned hot-line support

agent and she is required to provide them with the necessary information. Each of these groups is likely to have a different list of checkpoints, hence, from Mary's pool of information, she updates the groups with the most up-to-date news whenever possible, and only with those that are relevant to them at that point in time wherever they are in the midst of their tour. In our work, we will show how this can be done efficiently in a distributed computing environment where each node holds a list (similar to the checkpoint list for each group) that needs constant updating from a preassigned node.

1.3 Research Problems

In this section, we will discuss some of the problem issues raised in the computing arena. We will study some problems in query processing for different environments and later show how they can be addressed or alleviated using the proposed algorithms for managing caches. We begin through exploring ideas in a multi-user environment where multiple queries coexist together in the system. These queries usually access some common relations, or share some common subexpressions. To improve performance, intermediate/final results from earlier queries can be cached, and used to speed up evaluation of subsequent queries. Most of the existing caching strategies adopt a *predictive* mechanism to determine the results to cache: a result is cached if its *estimated* benefit is expected to be higher than its investment. Such strategies are effective when the query stream exhibits a high degree of locality for read-only applications (since updates will

result in cache invalidation). It unfortunately misses the dramatic performance improvements obtainable when the answers to a query, while not immediately available in the cache, can be obtained from concurrently running queries. Next we move to a Peer-to-Peer distributed environment where we observe that for unstructured query search, it can easily handle node insertions and deletions into the network without needing any centralized control, network management or data placement, however the overhead of flooding the network with messages means that it makes the system very unscalable. Hence, it is desirable to reduce the extend of this flooding without compromising overly on the effectiveness and efficiency of the search process. Moreover, implementation of the routing indexes usually requires exchanges of updates between nodes, and this will add on to the burden of the already heavily loaded network. Hence, it will be good to do away or at least minimize the amount of exchanges needed. In such a distributed environment, it can be more beneficial to work as a collaborated group than as an individual where sharing of commonly used data can help in reducing each others' workload. From our literature survey, we observe that most of the existing collaboration links are initiated by query activities, and feel that other ways of initiating a collaboration between nodes should also be explored. Lastly, we define edge computing to be able to push application logic and the underlying data into the cache to the edge of the network, with the aim of improving availability and scalability. However, the accuracy or the coherency of the data supplied to the applications depends on how efficiently updates can be disseminated to the edge servers, and in the literature, there are not a lot of work in this area. Hence,

maintaining the coherency in these caches with the source deserves closer examination.

1.4 Research Contributions

In this thesis, we have proposed a number of methods to improve the mechanism for processing queries using caches. For each method, the design and performance have been thoroughly described and examined, each complete with detailed experimental studies and analysis. We have carried out in-depth exploration with these methods and have shown that they can contribute significantly in improving the caching mechanisms in query processing.

In our first mechanism, we re-examine the issue of caching using a novel demand-driven caching framework, called *cache-on-demand* (CoD). CoD views intermediate/final answers of existing running queries as *virtual* caches that an incoming query can exploit. Those caches that are beneficial may then be materialized for the incoming query. Such an approach is essentially non-speculative: the exact cost of investment and the return on investment are known, and the cache is certain to be reused! We addressed several issues for CoD to be realized. We also propose three optimizing strategies: Conform-CoD, Scramble-CoD and Integrated-CoD. Conform-CoD and Scramble-CoD are based on a two-phase optimization framework, while Integrated-CoD operates in a single-phase framework. We conducted extensive performance study to evaluate the effectiveness of these algorithms. Our results show that all the CoD-based schemes can provide sub-

stantial performance improvement when compared with a predictive scheme and a no-caching scheme. Moreover, we show that Integrated-CoD offers the best performance but incurs the highest optimization overhead. Conform-CoD, which performs the worst in most cases, has the least optimization overhead. In addition, we have included several CoD extensions, to improve the overall performance of the query evaluation engine. It integrates three new techniques to realize this performance gain. The first method exploits intra-query parallelism where a sequence of operators within a query execution plan are executed in a pipeline. The second method explores the advantage of keeping multiple plans to increase the match space of CoD virtual caches at the expense of memory and comparison overhead. Lastly, the execution orders of plans may be reordered by the plan scheduler to further promote cache reuse.

Secondly, we propose CacheWire to instill collaboration among caches in a Peer-to-Peer(P2P) environment. We based our work on two observations on human behavior. First, when we need information, we usually ask our friends around us. Interestingly, our friends usually remember what we ask, and will come back to us if they need the same information later (knowing that we may have had the information since we have previously asked for them). Second, whenever we want to discard an item that is still usable (perhaps because we are clearing our office, moving, or have no need of the item anymore), we usually would pass it to a friend (or even charity organization) who have need of it. This prompted us to introduce the idea of a peer checking with its neighbors before it trashed out any cached objects. This is particularly beneficial if the cached

objects are computationally expensive to produce or the communication overhead may be high.

This architecture can be introduced to promote collaboration between network hosts in capitalizing their local knowledge to share their resources for answering its own as well as other URL/Query requests. It setup a framework that wires up available and willing network host/proxy caches qualitatively and constructively which allows hosts to become adaptive and community friendly. It is able to adjust its resources to keep items that are useful and at the same time share what it has learned with others. CacheWire supports decentralized collaboration between web proxy servers or application clients in a P2P environment. It learns, processes and remembers as it listens, and makes decisions that are based on the acquired knowledge. When a purpose arises, it handles it with the information at hand and makes constructive communication with others whenever necessary. Also, in our extensive performance study, we show that with selective collaboration it can obtain more answers at a lower communication overhead and response time.

Lastly, we propose two mechanisms for efficiently maintaining cache coherency. The first mechanism demarcates the local data set at each edge server, and another mechanism that identifies in logarithmic time those updates that apply to the local data set. The net result is that only updates to those portions of the database that are required by individual edge servers are propagated to satisfy their users and applications. The mechanisms work in conjunction with both eager and lazy update models. Analysis and experiment studies confirm that the proposed mechanisms can be very effective in

minimizing redundant updates to the edge servers.

1.5 Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 reviews some related works in the literature and also draws out the background knowledge necessary for building the proposed methods.

In Chapter 3, we discuss the Cache-On-Demand strategies in an environment where we have a *single central cache with data from a single or multiple sources*. This mechanism was realized in a centralized multi-user environment, we have proposed a novel method using demand-driven caching. Such an approach is essentially non-speculative: the exact cost of investment and the return on investment are known, and the cache is certain to be reused. Three different algorithms were proposed and evaluated: Conform-CoD, Scramble-CoD and Integrated-CoD. Also, we further extend the work on Cache-On-Demand in Chapter 4.

Next in Chapter 5, we present caching mechanisms for improving query routing and cache reuse in a Peer-to-Peer network where we have *multiple caches with data from multiple sources*. Here, we explored the caches for assisting query routing and answering. We proposed two strategies, one to promote the reuse of the cache content and another to improve the query response time through recall-routing. These two strategies were realized through salvaging cache-content and caching visiting-queries respectively.

We have conducted detailed experimental studies and evaluated these strategies in our work.

Chapter 6 presents an efficient updating method for the caches residing on the edge servers for maintaining *cache coherence for multiple caches with data from a single source*. These caches are located in a hierarchy of network servers where each server serves a cluster of users. two mechanisms were proposed and evaluated: data scoping and delta profiling. These mechanisms work in conjunction with both the eager and the lazy update models.

Finally, we conclude in Chapter 7 with directions for future research. The design and evaluation of the proposed mechanisms were also published in [37, 90, 38].

Chapter 2

Related Works

In this Chapter, some of the related works in the literature will be highlighted. First, it draws out the background knowledge necessary for building the proposed methods in Section 2.1. The background description covers a large range of issues on caching that include: the benefits, possible disadvantages, cache organization, types of cache objects, cache placement, cache management, environment, performance, desirable properties, coherency. Next, in Section 2.2, the related works corresponding to the proposed methods are categorized into their sub-Sections namely: Cache-On-Demand, CacheWire and Cache-Coherence. In these sub-Sections, more related works are referenced, highlighted and re-iterated for easy referencing.

2.1 Background

In [94], a survey on the different caching schemes was conducted. The highlights of the related points provide a good starting point for understanding the caching system. Caching has been recognized as one of the effective schemes to alleviate the service bottleneck and reduce the network traffic, thereby minimizing the user access latency. It is also used to deal with network congestion and server overloading. Clients within the firewall machine (at the proxy server) usually belong to the same organization and are likely to share common interests. This is so as clients probably access the same set of documents and tend to browse back and forth within a short period of time. Therefore on a proxy server, a previously requested and cached document would likely result in future hits. This saves network bandwidth and lower access latency for the clients.

Documents can be cached on the clients, the proxies, and the servers. The advantages of caching are to reduce bandwidth, reduce traffic, reduce congestion, reduce access latency by fetching from nearby proxy and making the transfer faster due to the less congestion (even for un-cached data), reduce workload of server, and provide extra availability. On the other hand, a list of possible disadvantages of caching are the presence of stale data due to lack of proper proxy updating, extra processing increase for cache miss, bottleneck occurring for single proxy cache, single point of failure for single proxy cache, and the reduction of hits to the original server.

For each client request, the required data will be retrieved from one of the following

locations: the browser's cache, the local proxy, the cooperative proxies or the server. The elements affecting the effectiveness and efficiency of caching comprises of the caching system architecture, the proxy placement, caching content, proxy cooperation, data sharing, cache resolution/routing, pre-fetching, cache placement and replacement, cache coherency, control information distribution, non-cacheable/dynamic data caching. In a cache system, there exists a number of desirable properties: fast access (access latency), robustness (availability), transparency, scalability, efficiency (any overhead, under-utilization of critical resources), adaptivity (adapt to user demands and changing network environment), stability, load balanced (without server bottleneck), ability to deal with heterogeneity (adapt to a range of network/hardware/software architecture), and simplicity (easier to implement and deploy).

In designing a caching architecture, it is presumed that when there is a bigger user community, there will be correspondingly higher probability that a cached document will soon be requested again. In all, there are three types of architectures: hierarchical, distributed, hybrid. For example, in hierarchical caching architecture, there can be made up of four levels: bottom, institutional, regional and national. During query lookup, we proceed in this order: the bottom, institutional, regional, national and finally if it is still not found, it has to lookup the original server. When the document is found, it travels down the hierarchy, leaving copy at each of the intermediate caches along its path. This kind of caching architecture provides more bandwidth efficient as proposed by the Harvest project [15] where popular web pages can be efficiently diffused toward the de-

mand. However, there are a few disadvantage in the hierarchical caching architecture, that is, it requires significant coordination among participating cache servers to place themselves at key access points in the network, each level may introduce additional delay, high level cache is bottleneck and long queuing delays, multiple copies of the same documents at different levels. Another form of architecture is the distributed caching architecture where there are only caches at the bottom level. First it may have a hierarchical meta-data-hierarchy to distribute directory information about the location of the documents and not distributing the actual document copies. The upper level caches are replaced by directory servers which contain location hints about the documents kept in every cache. Secondly, it can also use hashing for the caches to store only documents with URL that are hashed to it (CARP [93]). Thirdly, it can also use a central mapping service to tie together a certain number of caches (CRISP [34]). Lastly in this kind of architecture, the caches can inter-exchange messages indicating their content and keep local directories to facilitate finding documents in other caches (Summary Cache [32], Cache Digest [76], Relais project [72]). The final type of architecture is called the hybrid caching architecture where the caches may cooperate with other caches at the same level or at a higher level using distributed caching (ICP). Furthermore, it can also limit cooperation between neighbors to avoid obtaining documents from distant or slower caches, which could have been retrieved directly from the origin server at a lower cost. The survey indicates that the hierarchical caching architecture has a shorter connection time as compared to the distributed counterpart. It is also cited in [74] that the distributed

architecture has a shorter transmission times and higher bandwidth than the hierarchical type. Overall, the hybrid caching architecture combines the best of both the hierarchical and the distributed architectures.

Web caches are scattered all over internet, and it is important to be able to quickly locate a cache containing the desired document. Out-of-date cache routing information leads to cache misses. In order to minimize the cost of a cache miss, an ideal cache routing algorithm should route requests to the next proxy which is believed to contain the desired document and along (or close to) the path from the client towards the Web Server. When cache routing tables are used, like in Internet Cache Protocol (ICP), requests for web documents are forwarded up the hierarchy in search of a cached copy. In attempt to keep from overloading caches at the root, caches query their siblings before passing requests upwards. Otherwise, if hash function is used, like in the Cache Array Routing Protocol (CARP) which allows for ”queryless” distributed caching by using a hash function based upon the ”array membership list” and URL to provide the exact cache location of an object, or where it will be cached upon downloading from the internet. Another example of hashing cache technique is the Summary Cache where each proxy keeps a summary of the URLs of cached documents at each participating proxy and checks these summaries for potential hits before sending any queries. Summaries are stored as Bloom Filters [62] and updated only periodically. It reduces number of inter-cache protocol messages, bandwidth consumption and maintains almost the same hit ratio as ICP.

The hit rate is at most 40-50% regardless of caching scheme, so it's good to introduce the use of pre-fetching to improve the hit ratio. Pre-fetching anticipates future document request, preload or pre-fetch documents in a local cache. It reduces the client latency at the expense of increasing the network traffic. Pre-fetching can occur between the clients & servers, proxies & servers, or clients & proxies. Some pre-fetching techniques analyze the web server's reference patterns or the clients' web accesses. In the process of pre-fetching, the server pushes the popular documents to the proxies and in turn to the clients.

Another important aspect of caching is to be able to maintain cache coherence. Two different levels of coherencies are possible: strong or weak cache consistency. For strong cache consistency, the client validation method can be used, however this may incur too many unnecessary polling-every-time. Otherwise, server invalidation can also be used where the server keeps track of lists of clients to use for invalidating cached copies, however the lists themselves can be out-of-date. For weak cache consistency, we assume that the requesters are tolerant to the staleness of the requested page or we can assume that if a file has not been modified for a long time, it tends to stay unchanged. This method is being used in CERN httpd, Harvest cache. Another way to maintain weak cache consistency is to use the piggyback invalidation. Whenever a proxy cache has a reason to communicate with a server, it piggybacks a list of cached, potentially stale, resources from that server for validation. On the other hand, the server can also piggyback on a reply to a proxy, the list of resources that have changed since the last access by the proxy.

In the literature, a hybrid has also been proposed.

There are other forms of caching techniques, like using the proxy as a connection cache using persistent connections between clients and proxy and between proxy and web server, to improve the latency. The server can also migrate some of the services to proxies to alleviate server bottleneck to handle dynamic (non-cacheable) data. For hot spots at single server, it can be solved by using replication strategy to store copies of hot pages/services across several servers (proxies). In a proposed technique, the active-cache supports caching of dynamic documents at web proxies by allowing servers to supply cache applets to be attached to documents and requiring proxies to invoke cache applets upon cache hitting to finish the necessary processing without contacting the server, thus saving the bandwidth at the expense of CPU cost.

Briefly, we now discuss a simple set of classification for queries. First, there can be different levels of complexity for queries. Queries can be simple: it requires very little processing power and has low data usage (like as in Exact key-matching queries). On the other hand, queries can be complex: it requires a lot of processing power and has high data usage (like as in Multi-join queries, OLAP queries). Secondly, there can be different frequencies for query occurrence. That is, a particular query can occur just once, or on the other hand, many times repeatedly. Other categorizations of queries are also possible based on the input and the output characteristics. For instance, the input can be the required data resource for processing and the output can be the resulting data table. The input and output can itself be categorized and therefore be used to separate

these queries. The types of queries can sometimes determine the efficiency of deploying certain processing mechanism over another. In our work, we are interested to satisfy complex and repeated queries efficiently with the use of caches.

There are numerous work on caching in the literature. The development of the Internet and distributed systems has seen performance improvement through web caching as observed in [45, 32, 55, 48, 11, 20, 63, 51, 13, 92, 96]. A more recent interest can be seen in the area of web databases, where the mechanism of caching is specific to the database applications [5, 57, 50]. There are many different items that can be stored in the content for caches or replicas [46, 23, 35] and be reused by the same user or other users who query for these items. We can store a data resource fully (sometimes known as mirroring) or partially. It is especially beneficial to have the items nearer the inquisitor if the original data resource is situated far away from it, and retrieving the items is expensive and has a long delay. Answer or results for queries can also be stored for future reuse if computations of these queries are expensive, for instance queries involving the multi-join operator. Index paths can also be stored for future accesses in order to reduce the time needed to repeat the search for similar items within large indexes. Nodes belonging to these index trees can be cached as well, as in [19]. All in all, storing multiple copies of an item are particularly useful if queries requesting for these items are frequently repeated. On the other hand, it may not be useful if we know that these queries are simple, involving only local data accesses and occur just once.

We can classify caches into different categories. First, we can have a single central-

ized cache where all the queries will probe. Also, we can have multiple caches that are distributed and are either independent of each other or be cooperative [7, 80]. Secondly, these caches can either have items which are static (i.e. non-changing which are good candidates for mirroring to provide high availability and fault-tolerance) or they can hold dynamic values. For the latter case, caches usually opt to invalidate their cache values when changes are detected. Unlike caches, replicas are usually present in greater amount and require updates [65, 53, 10, 67] through the push or pull mechanisms.

In addition to work that focuses on what to cache and how to manipulate these caches, there are also work on how the data items can be stored and organized in these caches [2, 40]. On top of caching the content, in [4], a portion of the manipulating logic is also co-located at the remote data center or content delivery network like [3] which provides a distributed computing platform for edge computing.

In the next section, we will highlight some of the related works which we have investigated and found them relevant to the context of our work. We will then describe some of the related leading ideas and lay out some observations to distinguish our proposed methods as described in Section 1.3.

2.2 Related Works

2.2.1 Cache-On-Demand

Most of the existing work is done in the context of speculative caching strategies [16, 17, 26, 27, 28, 51, 81, 86]. These systems typically maintain statistical information on the usage of each cached result (e.g, when is the last use, what is the frequency of use, rate of use, etc.). A replacement metric can then be computed for the cached result based on the statistics. In this way, results with larger replacement metric may be removed from the cache, if necessary. These techniques also differ in the types of queries that are cached. For example, in [86], only the class of select-project-join-aggregate queries are considered, and only selection free select-project-join-aggregate queries can be cached.

A closely related area is multi-query optimization [25, 75, 79, 82, 91], where a set of queries are optimized collectively. In this way, common subexpressions need to be evaluated only once. The work in [82] looked at exhaustive algorithms that consider the common subexpressions during optimization. These techniques are too time consuming for "online" applications. Roy et al [79] proposed heuristic-based schemes for multi-query optimization. Given a set of queries, an AND-OR DAG representation is constructed. The sub-queries are represented as nodes in the DAG. Three search heuristic algorithms were proposed: Volcano-SH, Volcano-RU and Greedy. In general, these algorithms detect and exploit common subexpressions and determine which nodes to materialize.

In [89, 91], post-processing strategies are adopted, i.e., each query is optimized independently, and the sharing are considered only amongst the best plans of each query. These methods were shown to be inferior in generating globally optimal results [79].

Multi-query optimization has also been exploited to determine the set of intermediate as well as final results of queries to be cached [78] for subsequent queries.

As noted, multiple query optimization techniques are applicable only for queries that are submitted at the same time (or queries that are to be batched processed); otherwise, earlier queries will have to wait for subsequent queries before they can be optimized. More importantly, they are unable to exploit sharing that can be obtained from running queries.

2.2.2 CacheWire

In the first part of this section, we look at how pre-fetching helps in assisting in query retrieval. Our technique has similar benefit, however it does not pre-fetches the data but collects data as they passes through. In [55], a data pre-fetching strategy known as RAP was proposed. It predicts and pre-fetches data or documents by using a set of association rules identified from the web server's access log. In addition, their approach values recently added log records more than earlier log records. In contrast to using the hyper-link structure of the HTML document for prediction, they argued that the hyper-link structure may not be as relevant as the access log for learning the server's

access patterns. Further in [57], it employs a two-layer navigation model to capture both inter-site and intra-site access patterns, incorporated with a bottom-up prediction mechanism that exploits reference locality in proxy logs. In their work, they mentioned that models based on association rule mining are selective Markov models. A proxy server, sitting in the middle-tier in the internet infrastructure, serves many web-clients and covers a wide scale of the Web domain consisted of heterogeneous web-sites, and incrementally updating such models is a costly procedure. Such models are suitable only for applications where user access patterns are relatively stable. In their two-layer proposal, the first layer is a site-dependency graph, constructed to model navigation patterns among web-sites (SDG). In the second layer, it builds a page-level dependency graph, constructed for each web-site to model navigation patterns within an individual web-site. Next, [97] approach aims at clustering proxies which frequently access some web pages into information group. The number of web proxy servers on the internet can be very large in hundreds of thousands and their work focuses on efficiently maintaining and locating cache pages within these proxies. To locate cache pages, instead of using a fixed hierarchy that is formed statically, they proposed using proxy affinities to group proxies dynamically. Each proxy has a proxy profile that contains a list of frequently accessed URLs of web pages. That is, it keeps a log of the web page reference history of local clients, which is a sequence of web page references. The web pages are first partitioned into page clusters according to some reference patterns, formed according to the combined access patterns of all the proxies, where each proxy sends its profile

of local frequently accessed web pages, to a central web site, and an optimal or near optimal partition of frequently accesses web pages is generated. These resulting page clusters may contain overlapping web pages. From here, a proxy may choose to join an information group if it has an affinity for the corresponding cluster or if the intersection of the associated page cluster and the proxy's profile is significant. Their technique uses the Bloom Filter to encode the contents of page clusters as the content could be large. Finally, a central web site then selects a coordinator for each information group to co-ordinate the membership into the group. Each of these information groups should be of moderate size because a large group size incurs significant overhead for bookkeeping while a small group suffers from low cache hit ratio. A proxy joins enough information groups to cover its most frequently access pages. Their work aims to achieve little overhead and a relatively high cache hit ratio when they search for up-to-date version of the requested web page that is cached by some other nearby proxy. Cache discovery can be categorized to pull or push techniques. For pull, the proxy first finds which proxy caches the web page, and then retrieves the page. For push, when the content of the cache contents of a proxy changes, the proxy will tell other proxies about the change. Pull technique has high average response time, whereas push technique communicates a large amount of unnecessary information. In their work, they adopted a dynamic distributed collaborative caching infrastructure that is built according to the proxy affinities so that unnecessary communication between proxies of distant affinities can be avoided and changes of proxy affinities can be adapted to easily and quickly. They cited that the

majority of the user references go to only a small percentage of the web pages. In addition, there are many other techniques for mining web access pattern in the Web Mining literature like in [87, 98, 99].

Next, we look at some work on replication techniques and searches in P2P system which are relevant to our search algorithm design. [59] explores alternatives to Gnutella [36] search algorithms and data replication strategies. Gnutella is attractive for certain applications because they require no centralized directories and no precise control over network topology or data placement. However, Gnutella does not scale well as each query generates a large amount of traffic. Unstructured design is extremely resilient to nodes entering and leaving the system. However, the search mechanisms are extremely unscalable, generating large loads on the network participants. Replication means the number of nodes having a particular file, and replication ratio is represented as the percentage of nodes having the file. They have assumed a fixed network topology and a fixed query distribution as they mentioned that the time of search is short compared to the time of change in network topology and in query distribution, and that the results so obtained are still indicative of performance in a real system. In their report, they have listed many alternative parameters used in building the output performance metrics: network load, peer load, delays for positive answers, success rate, bandwidth of peer selection, fairness to both requester and provider. As in any flooding strategies, if TTL (the time-to-live or the number of hops/links where the the message is forwarded only up till so far in the network) is too low, the node might not find the object even

though a copy exists somewhere. They discussed two search methods, the expanding ring method and the random walk method. In the expanding ring method, a node starts a flood with small TTL, if the search is not successful, the node increases the TTL and starts another flood. It solves the TTL selection problem, but does not address the message duplication issue inherent in flooding. In the random walks method, to reduce the delay, they increase the number of walkers. With more walkers, they can find objects faster, but also generate more loads. They went on to suggest that 16 to 64 walkers will give good results. In random walks, there are two ways to terminate a walk, using the TTL limit, and performing checks with the original requester before walking to the next node. They observed through simulation that performing such checks is better, since there are a fixed number of walkers, having the walkers check back (improve by checking only after every 4 steps) with the requester will not lead to message implosion at the requester node. In Gnutella, owner replication is adopted, that is when a search is successful, the object is stored at the requester node only. Whereas in Freenet, path replication is adopted, that is when a search is successful, the object is stored at all the nodes along the path from the requester node to the provider node. They studied a third replication strategy known as the random replication, that is when a search is successful, they first count the number of nodes on the path between the requester and the provider, p, then randomly pick p of the nodes that the k walkers visited to replicate the object. Through simulation, they observed that in random replication, the performance is better than path replication except that it may be overly complex to implement. As for the

network topology, they have conducted simulation and found out that uniform random graphs are better for searching and data replication, as the high degree nodes in power-law random graph and the current Gnutella network bear too much higher load than average and introduce more duplication overhead in searches.

Squirrel [45] proposed a decentralized web caching algorithm to enable web browsers on desktop machines to share their local caches. It uses a routing substrate called Pastry [77] to identify and route to hosts that cache copies of a requested object. Squirrel makes use of the browser cache and uses the URL as the key of the object cached. A remote host stores either the actual object or maintains a directory of information about a small set of hosts that store the object. All objects stored in each host's cache are treated equally by the cache replacement policy regardless of whether it is its own or received data. Squirrel primarily deals with the problem of locating objects directly found in other caches or locating hosts that store these objects. We share similar interest and have considered several extensive hosts collaboration strategies. In particular, information is gathered from received request/data and processed toward not only benefiting self but others as well. In addition, we look at how caches from different hosts can collaborate to improve their fullest utilization before being flushed.

P2P technology has been popularized by Napster[64], ICQ[43], Freenet[21], Gnutella [36], Kazaa[49] etc. for file exchanges and instant messaging services. Most of the P2P network is seen as a collection of resources that are owned and managed by the users. Most of the users or peer nodes connect to one another directly without a centralized con-

trol point (some systems, e.g., Napster, use a group of servers as coordinators). In the P2P context, existing data retrieving techniques can be broadly categorized into DHT-based systems and flooding-based systems. While DHT-based systems provide a guarantee in performance (typically, logarithmic in the number of peers), the effort to maintain the systems is significant. Flooding-based systems, on the other hand, incur significant overhead in querying (as messages have to be broadcast from a node to its neighbors, which will then forward to other neighbors), but has little need for maintenance of meta-data across nodes. In this thesis, we follow similar ideas on flooding-based systems, and examine how the overhead can be reduced.

In a non-centralized querying system where data and queries are distributed across many nodes, caching is not the only factor that reduces the response time for answering a query. Given a query, there is also a need to propagate the query and locate the results in the network efficiently. As mentioned, there are two main categories of schemes to propagate queries in P2P systems. In the flooding-based schemes, queries are forwarded from a node to other nodes. These systems require minimum interaction among nodes. Example of these systems include Freenet[21] Gnutella [36], and OpenCola [66]. These systems adaptively cache information on nodes as necessary to meet demand regardless of what the node has downloaded. The availability at many nodes allows requested files to be downloaded from the nearest node rather than the original source. In our study, we share similar vision with these systems and aim to provide a general object-sharing framework based on object and query caching. The second category of schemes is based

on distributed hash table (DHT) [70, 71, 100, 88, 61, 33]. In these schemes, each node maintains a routing table that enables a query to be routed from node to node where each route is moving toward the node that contains the content. As such, there is a guarantee in performance (typically the number of hops required is of the order that is logarithmic to the number of nodes in the system). However, the dynamic nature of the network led to frequent updates to be made to these routing tables. In our work, we aim to minimize these updating procedures by passively storing information that passes through rather then actively gathering information by involving other nodes.

Next, we discuss two routing strategies as proposed in the literature. First, instead of selecting neighbors at random or by flooding the network by forwarding the query to all neighbors, [24] proposed a method to forward only to a set of neighbors that are more likely to have answers. This method requires the nodes to exchange routing information in order to assist in the forwarding of queries to the set of selected neighbors. The information contains the number of documents along each path and the number of documents on each ”topic” of interest. Queries are made up of conjunction of subject topics where documents can have more than 1 topic and document topics are independent. Whenever there is a query that cannot be satisfied by the local node, the routing information is consulted and it returns the direction of search where it contains the most wanted data. Each node has all the neighbors' statistics that comprises a list of accumulated number of documents on each topic. To maintain the freshness of this routing information, neighbors will have to exchange details for each neighboring connection,

disconnection and data updates. Furthermore, this updated information has to be propagated to all linked neighbors and when there is a need to handle more topics, a larger node space will be required. In CacheWire, the nodes build their routing tables based on the query visits, hence the propagation of information to maintain the routing table is not needed. Moreover, caches are used to provide routing information instead of a statically mapped tables.

In the second example, we look at Freenet [21] which describes another form of routing strategy. Each node contains a routing table that holds binary file keys representing the queries that passed by and information on the data sources. A successful search will update all the nodes along the path with the source information and at the same time copy the data into their caches. Hence, any subsequent request for the same key will be immediately satisfied from the local cache. This strategy is built upon the willingness of the nodes to cooperate through space donation for storing replicated files in the caches even though they are not beneficial to the local nodes. In our work, we do not look further after the query has passed the node, hence there is no need to cache any results from successful queries belonging to other nodes.

In the database arena, for example in SQL Server 2000, it allows a kind of caching known as Snapshot replication. In snapshot replication, data is distributed to the remote users in its original form at a specific time. An example Snapshot can consist of small tables. The data can be updated, however, the entire updated Snapshot will have to be transmitted. Oracle 8i too has Snapshot capabilities. It is also known as materialized

views in Oracle. In CacheWire, our focus is to cache queries which act as direction pointers to the data that are being requested and not on making copies of the data. Also, CacheWire uses Data-Centric Routing, in that it looks at the named data concerned and aims to avoid doing unnecessary routing and forward the request only to the nodes leading to the destination.

Caching has also been proposed and implemented in a data warehousing system, in particular, a OLAP systems which aims at reducing the response time. In [81], a data warehouse cache manager was presented which caches the query results together with the query string. The cache replacement and admission algorithms make use of a profit metric, which considers the average rate of reference, its size, and the execution cost of the associated query. In [28], a chunk-based caching approach was proposed. It allows queries to partially reuse the results of previous queries which they overlap and chunk miss is handled by a new physical organization for relational tables known as chunked files. Another cache manager was proposed in [52]. It uses multidimensional range fragments as the basic logical unit which provides a finer granularity for materialization. In [47], an OLAP query caching approach in a P2P network was proposed. It constructs a large virtual cache by sharing the content of individual caches and works toward benefiting all peers. A voluntary caching policy was proposed. The caching policy attempts to exploit under-utilized resources that may exist in some peers and at the same time avoid wasting any result that has been obtained from the warehouse. When a peer with a full cache is unable to insert another entry with benefit lower than any of the entries in

the cache, it will ask whether any neighbor wants to cache it. In CacheWire, the cache salvaging policy differs with this policy in many aspects. First, the item to be given away is from the cache and not a new entry. Secondly, each item to be given away has an accompanied trend information meant to assist neighbors to make item-acceptance decision. Thirdly, we do not just give the item to all neighbors but only to neighbors that potentially have use for them.

2.2.3 Cache-Coherence

Data replication is supported in DBMS like Microsoft SQL Server and Oracle to improve performance and fault-tolerance. By default, many of those DBMSs implement the lazy update model [54, 9, 8] where changes made by a transaction are applied to the other replicas only after the transaction commits. While the synchronization overhead is low, lazy update could lead to inconsistency among the replicas. The alternative is eager replication, where all the replicas are synchronized as part of an atomic transaction. Gray et al [39] proved that, with eager replication, a ten-fold increase in number of replicas could push up deadlock occurrences by a thousand-fold. It discusses some of the dangers of replication. In eager replication, it keeps all replicas exactly synchronized at all nodes by updating all the replicas as part of one atomic transaction. Eager replication reduces update performance and increases transaction response times because extra updates and messages are added to the transaction. It is mentioned that mobile appli-

cations require lazy replication algorithms as most nodes are normally disconnected. It asynchronously propagates replica updates to other nodes after the updating transaction commits. Lazy replication has its shortcomings, the most serious being stale data versions. Eager replication delays or aborts an uncommitted transaction if committing it would violate serialization. However, Lazy replication has a more difficult task because when the serialization problem is first detected, there is usually no automatic way to reverse the committed replica updates, rather a program or person must reconcile conflicting transactions. There are two ways to regulate replica updates: Group and Master. For group updates, any node with a copy of a data item can update it, this is called update anywhere. For master update, each object has a master node, and only the master can update the primary copy of the object, where all other replicas are read-only. Replicating data at many nodes and letting anyone update the data is problematic, as security and performance are issues to handle. With a transactional model, Eager with a high transactional rate means dramatically higher deadlock. Whereas, Lazy-group just converts waits and deadlocks into reconciliations. Generally, the master scheme performs better but still suffers from increased deadlock as the replication degree rises. Also, none of the master schemes allow mobile computers to update the databases while disconnected from the system.

However, more recent work such as [95] have shown that eager replication could be viable when implemented on group communication primitives.

The other related area, data caching, has also been researched into extensively. The

purpose here is to place read-only copies of data objects near user applications to satisfy their queries. This technique is widely used to cache web objects; [6] and [94] provide comprehensive surveys on web caching. The web content that are cached could range from static pages, to dynamically generated objects (e.g. see [14]). More recent work by Shah et al [83, 84] addressed how to organize data repositories in a dissemination tree according to their coherence requirements on a time-varying dataset, such that only data changes that exceed the threshold allowed by the coherence requirement of a branch need to be propagated down. This requires a parent server to track the data values at each of its dependent servers in order to detect when the coherence requirement is violated.

There are many works in dealing with cache updates. [10] introduces latency-recency profiles, a set of parameters that allow clients to express preferences for their different applications. A cache or portal uses profiles to determine whether to deliver a cached object to the client or to download a fresh object from the remote server. Their work is based on the observation that clients may have different preferences for the latency and recency of their data, some prefer the most recent data, others will accept stale cached data that can be delivered quickly. They reported that techniques used to keep cached objects up-to-date cannot handle diverse client preferences and may perform poorly with respect to either latency, recency or bandwidth consumption. Cached data becomes stale as updates are made at remote servers, hence there is need for techniques to keep cached data consistent with data at the remote servers. They define TTL as the estimated length of time the object will remain fresh. Upon expiration, validation is needed and this

generates overhead. However, it is difficult to estimate accurately an object's TTL. The most commonly used mechanisms in proxy caches to maintain cache consistency are: pre-fetching and server side invalidation (SSI). Pre-fetching occurs in the background and keeps cached objects up to date. However, it consumes large amount of bandwidth and does not scale well to large number of objects. AUC, always-use-cache is commonly used by web portals that maintain copies of objects from many web sites. The second technique, SSI, servers maintain information about objects stored in client caches, and send invalidation messages to caches when an object is updated. In their proposal, they define profiles which comprise of a set of parameters that allow clients to express their preferences with respect to latency and recency for their different application. Profile is a generalization of TTL and AUC. The parameters can be tuned to provide performance anywhere between these two extremes. They have also discussed some other work that allows cached copies of objects to deviate from the data at the server in a controlled way. This requires servers to propagate updates to the client-side cache when a cached value no longer has an acceptable degree of precision; which places a burden on servers and does not scale well to a large number of clients. In their work, they require no such cooperation from the remote web server.

Furthermore, [65] proposed an algorithm to minimize the overall divergence between source objects and cached copies by selectively refreshing modified objects. Ideally, cached copies of data objects are kept transactionally consistent with the source copies at all times. The propagation of updates may be infeasible as data collections may be

large or frequently updated, and network or computational resources may be limited. In most refresh scheduling policies, the cache plays the central role: refreshes are scheduled entirely by the cache and implemented by polling the sources, without sources participating in the scheduling. However, improvement to synchronization can be achieved through some level of source participation: source can give priority to servicing local user queries as they occur and participate in cache synchronizing with any spare bandwidth. Also, source can have a say in weights given to different data objects. In their work, they focus on stale caching environments with a large number of sources that synchronize their data with a shared cache. The value of the object differs at the source and cache is known as divergence. To measure the divergence, there are 3 methods, namely Boolean freshness (up-to-date or not), number of changes since refresh, and value deviation. They observed that refreshes based solely on the weighted divergence does not generally lead to good refresh schedule. In their work, they have established a priority policy that achieved much better synchronization. Typically, sources are not aware of the state of the content of other sources. However, ideally, all modified objects having priority above a global threshold T should be refreshed, but maintaining such global T is infeasible. So, each source must maintain its own independent copy of the refresh threshold, and some protocol to loosely regulating the thresholds. The proposed technique relies on occasional feedback messages from the cache requesting that sources raise or lower their thresholds. They had assumed that the time required to propagate a modified object from a source to the cache is small enough to be neglected. They allow

the tuning of weights used, so certain important objects are refreshed more aggressively than others. The cache sends positive feedback messages when the refresh rate is too slow, asking sources to decrease their thresholds and thereby increase the overall refresh rate. In the absence of feedback, sources can assume that the refresh rate is too fast and should reduce the refresh rate by increasing their thresholds. Also, the cache continuously monitors cache-side bandwidth utilization. If underutilized, the cache uses the excess bandwidth to send positive feedback messages to as many sources as possible. Sources with the highest local thresholds are selected first. Each source can piggyback its current threshold in refresh messages. In their work, they have shown that source cooperation in the synchronization process is advantageous.

Next, [56] has conducted experiments to show that a significant part of the query workload can be offloaded to cache servers, resulting in greatly improved scale-out on the read-dominated workloads. A multi-tier environment comprises of three components: the browser-based clients, mid-tier application servers and a backend database server. To improve performance and scalability, it must reduce the load on the backend server or increase the capacity. The goal of the mid-tier database caching is to transfer some of the load from the backend database server to intermediate database servers. Also, applications should not be aware of what is cached and should not be responsible for routing requests to the cache or the backend server, as one of the key distinctions between caching and replication is that the process should be transparent to the applications. In their work, they define their MTCache server to contain empty tables but

maintaining the table statistics, indexes and materialized views to reflect the data on the backend server. All the queries are submitted to the MTCache whose optimizer decides whether to compute a query locally, remotely or part locally and part remotely. The cache server allows caching of horizontal and vertical subsets of tables and materialized views. Moreover, their approach is not tied to replication updating techniques as it works equally well with any other (synchronous or asynchronous) mechanism for update propagation. For distributed queries, a query can access tables on one of more linked servers and the local server. Currently, query optimization is largely heuristic. Their implementation for mid-tier database caching relies on SQL server support. Distributed transactions are supported on SQL server where an application can, for example, update some local data and data on several linked servers all within a single (distributed) transaction. SQL server replication is based on a publish-subscribe paradigm. Briefly, a publisher (source) makes data available for replication through a distributor (middleman) that propagates changes to subscribers (targets). A publisher defines what data it allows to be replicated by creating one or more publications consisting of a set of articles. An article is defined by a select-project expression over a table or a materialized view. Changes to a published table or view are collected by log sniffing and inserts them into a distribution database on the distributor. The distributor is responsible for propagating the changes found in its distribution database to the affected subscribers. Once changes have been propagated to all subscribers, they are deleted from the distribution database. Propagation of changes to a particular subscriber can be initiated either by the subscriber

(a pull subscription) or by the distributor (a push subscription). The propagation is performed by a separate agent process that wakes up periodically, checks for changes and, if there are any, applies them. The difference is whether the agent runs on the subscriber machine or the distribution machine. Suppose the backend database server has become overloaded, resulting in poor response times. The goal is to switch some of the load to smaller and cheaper intermediate cache servers (running SQL Server) without having to modify applications in any way. All the shadow tables (in MTCache), indexes and materialized views are empty. However, all statistics on the shadow tables, indexes and materialized views reflect their state on the backend database. The shadowed statistics are needed for query optimization. The data actually stored in the MTCache database is a subset of the data from the backend database. The subset consists of materialized select-project views of tables or materialized views on the backend server. What data to cache is user controlled. The cache data can be thought as a collection of distributed materialized views that are transactionally consistent but may be slightly out of date. A cache server may store data from multiple backend servers. To enable the cache: it must first configure by specifying which machines will act as publishers, distributors, and subscribers. The DBA then must decide what data to cache. To cause an application to connect to the mid-tier server instead of the backend server, there is only a need to redirect the appropriate ODBC sources from the backend server to the mid-tier server. There are two forms of optimizations, remote and local. In remote optimization, each user sends the sub-expression to the backend server and have it return the estimated cost

and cardinality. This method is not preferred as it has a high expected overhead. In local optimization, the required catalog information and statistics are replicated on the local server and the sub-expression is optimized locally. The motivation for evaluating it locally is that, even though the backend server may be powerful, it is likely to be heavily loaded so we will only get a fraction of its capacity. In their work, they have presented a prototype of the mid-tier database caching can improve the system throughput and response time by transparently offloading some of the query processing load from the backend database server to cheaper cache servers.

The primary distinction of our work is that we aim to minimize data dissemination by updating only the *active* dataset on each repository, i.e., the portion of the database that is queried by the local applications. As this approach is not exploited by the existing work on data replication and cache described above, our proposed solution complements them rather than replaces them. For example, it is possible to apply our data scoping and delta profiling techniques in the framework of [83, 84]. We will also show in this thesis how our techniques can work together with both eager and lazy update models.

Chapter 3

Cache-On-Demand

3.1 Introduction

In a multi-user environment, computing resources are usually being stretched to their limits. Hence, efficient algorithms designed for applications are very much in demand. In the context of database processing, one of the crucial components is the query processor. The design of such processor is inclined toward better optimization and evaluation techniques without incurring excessive overheads.

Queries in a multi-user environment coexist together in the system. For any instance of the execution status of a query retrieval system, there are three categories of queries: queries that have been evaluated completely, queries that are partially evaluated, and queries that are awaiting in a queue for admission to be evaluated. Moreover, queries posed to a database usually access some common relations, or share some common

sub-expressions. To exploit the common sub-expressions, one approach is to optimize a set of queries collectively. However, in most applications, queries are evaluated as they arrive, and are not batch-processed. Moreover, batching such queries translates to longer waiting time for queries that arrive early. In other words, *multi-query optimization* techniques [25, 75, 79, 82, 91] cannot be directly applied to these context.

An alternative and more promising approach is to cache intermediate/final results from earlier queries. The cached data that match the sub-expressions of subsequent queries can then be used to speed up their evaluation. However, most of the existing caching strategies adopt a *speculative* mechanism to determine the results to cache [16, 17, 26, 27, 28, 51, 81, 86]. Under the speculative schemes, we are actually working with the first two categories of queries: completed queries' results (intermediate or final) are cached to be reused by running queries. A result is cached if the estimated benefits of reusing the cache is higher than the investment (i.e., the cost incurred to cache it). We note that while the cost of caching the result can be determined, the return on investment depends on future queries that use the result and is typically estimated based on histories of past queries. Because of the element of uncertainty on the usefulness of the cached data, such strategies are effective only when the query stream exhibits a high degree of locality. In addition, the cache will have to be invalidated whenever updates occur, rendering such methods useful only for read-only applications.

More importantly, existing predictive caching approaches miss the dramatic performance improvements obtainable when the answers to a query or its sub-expressions,

while not immediately available in the cache, can be obtained from concurrently running queries. In this chapter, we re-examine the issue of caching using a novel demand-driven caching framework, called *cache-on-demand* (CoD). CoD exploits partially evaluated queries for the waiting queries, i.e., it looks at intermediate/final answers of existing running queries as *virtual* caches that an incoming query can exploit. Those virtual caches that are beneficial may then be materialized for the incoming query. Since the exact cost of investment and the return on investment are known, the benefit is certain and guaranteed. In other words, such an approach reuses results with *certainty* of reuse. We note that the cost of investment may be zero if the query actually stores the target cache as an intermediate result for the rest of its query (especially for multi-relation queries).

To realize CoD, several issues have to be addressed. First, given an incoming query, we need to identify candidate virtual caches to be materialized. Second, since running queries may have already produced some of their answers, some of the candidate virtual caches may be "incomplete". This calls for a mechanism to handle the remaining missing portion. Third, we also need a mechanism to "alter" the plans of running queries to materialize the virtual cache. Finally, for an arriving query, there may be more than one combination of virtual caches to be exploited. Picking the "optimal" candidate virtual caches efficiently (and with minimum overhead) is critical. We shall address these issues in this chapter.

In this chapter, we propose two categories of caching strategies. The first category adopts a two-phase approach for the CoD framework. In the first phase, the incoming

query will be optimized without considering the currently running queries. In the second phase, the plan is postprocessed to exploit any virtual caches that are beneficial. Such a strategy allows us to reuse existing optimizer and simplifies the problem to a manageable level. We also propose two optimizing strategies that are based on the two-phase CoD framework: Conform-CoD and Scramble-CoD. The two strategies differ in the second phase. While Conform-CoD preserves the ordering of the plan from phase 1, Scramble-CoD may "scramble" the plan to exploit more common sub-expressions.

The second category essentially integrates the two phases of the first category into a single phase. In other words, the plan of the incoming query is generated by considering materialization of the virtual caches. Such an approach is expected to produce better quality plans than the two-phase schemes. However, the runtime overhead may be higher. Moreover, it may require a redesign of existing optimizer. We propose an algorithm called Integrated-CoD in this category.

We conducted an extensive study to evaluate the performance of the three strategies against two other schemes: a predictive scheme and a no-caching strategy. Our results show that the CoD-based schemes can provide substantial performance improvement over the predictive and no-caching schemes. For the two-phase schemes, Scramble-CoD outperforms Conform-CoD in most cases, but Integrated-CoD offers the best performance at the highest optimization overhead.

To reduce the optimization overhead, we also adopted the simple heuristic of restricting the maximum number of relations that can be cached. This is intuitive and reasonable

since common sub-expressions are more likely to be common among a small number of relations than a large number of relations. We apply this heuristic on all CoD-based schemes that we study in this chapter. Our further performance study showed that such a heuristic can cut down the optimization overhead without sacrificing much on the quality of the plans.

The remaining of this chapter is organized as follows. In the next section, we present the cache-on-demand framework. Section 3.3 presents the two categories of optimization strategies. In Section 3.4, we report our experimental study and results. Finally, in Section 3.5, we summarize the 3 CoD strategies which we have proposed so far.

3.2 Cache-On-Demand

In this section, we shall first present the basic framework for cache-on-demand. Next, we shall discuss our solutions to three of the issues. We shall also present the system architecture.

3.2.1 The Big Picture

The basic idea behind cache-on-demand is simple: cache only those beneficial results that are certain to be reused. To be certain of reuse, instead of *predicting the future* (as is done in traditional caching strategies), we focus on the *present* where we have "perfect" knowledge of what is happening in the system. In other words, instead of asking "What

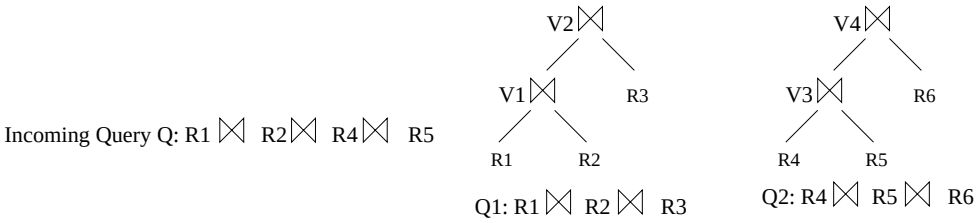
to cache so that future queries may benefit?", we ask for each incoming query "What to cache from the existing running queries that the incoming query (or rather the system) may benefit?"

Essentially, under cache-on-demand, the incoming query views the set of (currently) running queries as potential candidates to provide results – intermediate or final – that it (i.e., the incoming query) can reuse [1]. As such, we can view the intermediate/final results of running queries as *virtual caches* that are only materialized for reuse if they are useful to the incoming query. In this way, there is also a *certainty* of cache reuse (unlike speculative techniques where a cache may be replaced before reuse).

Thus, the system operates as follows. When a query arrives, the system (1) examines the queries that are currently running in the system, determines the common subexpressions between the incoming query and the running queries, and identifies the intermediate/final results of running queries that can be reused for the incoming query; and (2) materializes these results for the incoming query. To illustrate the concept, consider the example shown in Figure 3.1. Here, we have an incoming query Q which is a four-way join: $R_1 \bowtie R_2 \bowtie R_4 \bowtie R_5$, and there are two running queries: $R_1 \bowtie R_2 \bowtie R_3$ and $R_4 \bowtie R_5 \bowtie R_6$. Clearly, caching the results of $V_1 = (R_1 \bowtie R_2)$ and $V_3 = (R_4 \bowtie R_5)$ will be useful to Q (assuming that it is beneficial overall). Since we can determine more certainly the cost of investment (to cache V_1 and V_3) and the return from the investment

[1] We note that, in this chapter, we are looking at reusing *complete* answers of (join) operations (intermediate or final operations in the query plan). However, as we shall see, not the complete results need to be generated at one go (see Issue 2 in this section).

from Q, we can determine if caching them are beneficial easily.



(a) Incoming query Q (b) Two queries running in the system

Figure 3.1: Example to illustrate cache-on-demand.

To realize Cache-on-Demand, we have identified several issues that have to be addressed:

1. Given an arriving query, we need a mechanism to determine the virtual caches that are relevant to the query quickly.

2. Since running queries may have already been partially evaluated when the new query arrives, some of the candidate virtual caches are effectively "incomplete" (as some answers may have already been produced but are not cached). When these virtual caches are to be materialized, we need a mechanism to determine what can be cached and what has to be re-evaluated.

3. For virtual caches that have to be materialized, we need a mechanism to inform the associated running queries to cache them. Such a mechanism should not be too disruptive to the running queries.

4. For a given query, there may be more than one combination of candidate virtual caches that can be exploited. For example, a query $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ matches plans $P_1 = ((R_1 \bowtie R_2) \bowtie R_3)$ and $P_2 = ((R_2 \bowtie R_3) \bowtie R_4)$ in their virtual caches. To determine the "optimal" candidate virtual caches to be used, we need to develop efficient novel optimization algorithms.

We shall address the first three issues in the next few subsections, and leave the discussion on the last issue to the next section. For simplicity, we made several reasonable assumptions. First, we shall restrict our work to multi-join queries (without restrictions and projections). Restrictions can be dealt with easily by treating the restricted relation as a "new" base relation. Second, we consider a virtual cache matches a query subexpression only if it contains the answers to the subexpression.

3.2.2 Issue 1: Finding Candidate Virtual Caches

The first issue essentially calls for information on virtual caches to be available. In this work, we employ a data structure called the VirtualCache (VC) table to keep track of the information on virtual caches of all running queries. Recall that a virtual cache corresponds to a result, intermediate or final, of a running query. Referring to Figure 3.1, the system would have the following virtual caches: for Q_1, we have $R_1 \bowtie R_2$ and $R_1 \bowtie R_2 \bowtie R_3$; and for Q_2, we have $R_4 \bowtie R_5$, and $R_4 \bowtie R_5 \bowtie R_6$. The information maintained include the relational algebraic (RA) expression of the virtual cache, a

counter on the number of queries sharing this virtual cache, whether the cache is still valid for subsequent queries (in case the participating relations have been updated), etc. Essentially, virtual caches are added whenever a new query is admitted for execution, and the virtual caches may be deleted whenever all queries sharing the same virtual caches have been evaluated (i.e., counter = 0).

To facilitate quick lookup, these information are organized into a hash table based on the relation ids and the join attributes. In this way, $R_1 \bowtie_{R1.A=R2.B} R_2$ and $R_1 \bowtie_{R1.A=R2.C} R_2$ will be located in different partitions. In the case that restrictions are considered, we may have $\sigma_{R_1.A>5}(R_1) \bowtie_{R1.A=R2.C} R_2$ being hashed to the same location as $\sigma_{R_1.A>10}(R_1) \bowtie_{R1.A=R2.C} R_2$. In this case, all matching virtual caches may be examined and the most beneficial one will be picked.

3.2.3 Issue 2: Salvaging the Virtual Cache

Suppose the arriving query and a currently running query share a common join $J = R_1 \bowtie R_2$ in their plans. We shall refer to the arriving query plan as Q and the plan with the candidate virtual cache as P. In the following discussion, we shall focus on what can be salvaged, and ignore the issue of whether the cache is beneficial. The issue of beneficial cache shall be addressed in the next section.

We shall also focus on three join algorithms: GRACE hash join, sort-merge join, and nested-block join. Note that what is important is the join method for P; the join method

for Q is "not" as critical: if the cache can be salvaged from P, then Q's join method is irrelevant anyway. There are two cases to consider. First, P has not produced any answer tuples of J yet. There are two cases for this scenario: (1) P has not begun evaluating J yet; (2) P has not begun the join phase yet – for hash join, P may be partitioning the relations participating in J; for sort-merge join, P may be sorting the participating relations. Clearly, the results of J in P can be completely salvaged and cached for Q (regardless of the join method adopted in P). Second, P has begun evaluation of J and has partially produced some answer tuples. Depending on the join methods adopted by P, different degree of salvaging is possible:

- *GRACE hash join.* Suppose P splits the join into m subjoins, i.e., $J = \cup_{j=1}^{m} R_{1j} \bowtie R_{2j}$ and $R_i = \cup_{j=1}^{m} R_{ij}$ for $i = 1, 2$. Under the GRACE hash join, if P has partially evaluated J, it means that some of the m subjoins have been evaluated, and the results are not cached [2]. Suppose subjoins 1 to k ($k < m$) have been evaluated. In this case, Q can reuse the following: the virtual caches for subjoins $k+1$ to m, and the partitions of R_1 and R_2 for subjoins 1 to k. In other words, Q need to evaluate the subjoins 1 to k again. [3] There is, however, one complication: if the memory size allocated for P is larger than that for Q, then, it is possible that the partitions generated from P may not fit into the memory for Q to evaluate them. In this case, Q may have to further split each partition. This is still cheaper

[2] We assume here that in P, it is not required to cache J. Clearly, if P already decides to cache J, for subsequent operations within P, then this step is not needed.

[3] We assume that the partitions are not removed until the entire join is completed.

than reading the original relation. In the worst case, all the partitions have been evaluated; here, the cost is the same as reading the original relation.

- *Sort-merge join.* P is considered to have partially evaluated J only during the merge phase. We split the merge phase into a sequence of merge steps (say m), each of which merges a certain range, i.e., $J = \cup_{j=1}^{m} R_{1j} \bowtie R_{2j}$ where R_{1j} and R_{2j} are the jth (logical) range partitions of sorted R_1 and R_2 respectively, and $R_i = \cup_{j=1}^{m} R_{ij}$ for $i = 1,\ 2$. We note that this partitioning is a logical one only. Thus, like the GRACE hash join approach, if some of the partitions have been merged, we can reevaluate them, and salvage the remaining result tuples. Note that since the file is physically sorted the reevaluation process is done for partitions at the beginning of the relations. In this sense, there is no additional overhead than what is necessary to produce the answers.

- *Block-nested loops join.* When block-nested loops join is used, the same logic applies. Essentially, the outer relation, say R_1, is logically partitioned into fragments, where each fragment is the portion of R_1 that fits into the memory that corresponds to one iteration of the algorithm. Thus, we only need to evaluate the join between fragments of R_1 that cannot be salvaged against R_2. Like the GRACE hash join, care has to be taken should the memory size allocated for P is larger than that for Q. In this case, k_p fragments of P can be split into k_q fragments of Q where $k_p < k_q$.

3.2.4 Issue 3: Synchronization between the Incoming Query and the Running Queries

The incoming query needs to know the running status of the virtual cache in order to know what to salvage. Similarly, a running query needs to be informed that it has to cache its intermediate result (if it benefits the incoming query). To do so, we employ another data structure which we referred to as the OperationExecutionStatus (OES) table. OES keeps the execution status of the virtual cache (or operations) that are running. For example, for a hash join operation, OES keeps track of the total number of subjoins, the subjoin relations, for each subjoin a flag that indicates whether it has been evaluated, the addresses of the relations and the intermediate results (if they are cached), status flag etc.

For an incoming query, when it finds that a candidate virtual cache has not been materialized and that the operation corresponding to the virtual cache is currently being evaluated [4], the incoming query must also need to look up OES to see if the cache has been partially evaluated and what can be salvaged. In addition, whenever such a virtual cache can be salvaged, the incoming query will update the status flag for each subjoin operation to indicate that they have to be cached. In this way, the running query can be made known to cache the subsequent subjoin operations.

On the other hand, for a running query, the evaluation of its operation is done by "collecting" the sub-operations from the OES. In other words, whenever the execution

[4] These information are obtained from the VC table.

engine evaluates a sub-operation, it checks OES to see if it needs to be cached. If the status flag is set, then the sub-operation is evaluated and its result is cached. Thus, we can see that the proposed mechanism minimizes disruption to the running queries.

3.2.5 The "Goodness" of a Virtual Cache

Given that we have determined what can be salvaged, we have to determine whether the virtual cache is beneficial. We have adopted a cost-based approach for this purpose. This turns out to be fairly straightforward since we know the cost of investment $\mathcal{I}$ (i.e., to materialize the virtual cache), the cost of the original query plan for the arriving query (without any virtual caches) $\mathcal{O}$, and the cost of the refined query plan using the virtual caches $\mathcal{R}$. The benefits, $\mathcal{B}$, is given as follows:

$$\mathcal{B} = (\mathcal{O} - \mathcal{R}) - \mathcal{I}$$

We note that $(\mathcal{O} - \mathcal{R})$ corresponds to the return on investment from the virtual cache. If $\mathcal{B} > 0$, we say that it is a beneficial virtual cache, and it is worth materializing the cache. Otherwise, using the virtual cache will lead to overall poorer performance for the system: the running query incurs a higher cost to materialize the virtual cache, but the gain for the arriving query is much smaller. We also note that it is possible for $\mathcal{I}$ to be zero in many cases: the virtual cache has already been materialized by another query or the original query (that needs it to be materialized for subsequent operations of the query).

3.2.6 The System Architecture

Figure 3.2 sketches the architecture to support cache-on-demand. We assume a process-per-user architecture where each user-session has a separate process serving its queries. The optimizer process optimizes the incoming query and exploits any beneficial virtual caches. In addition, a *scheduler* process determines when a (sub)query process is ready for execution.

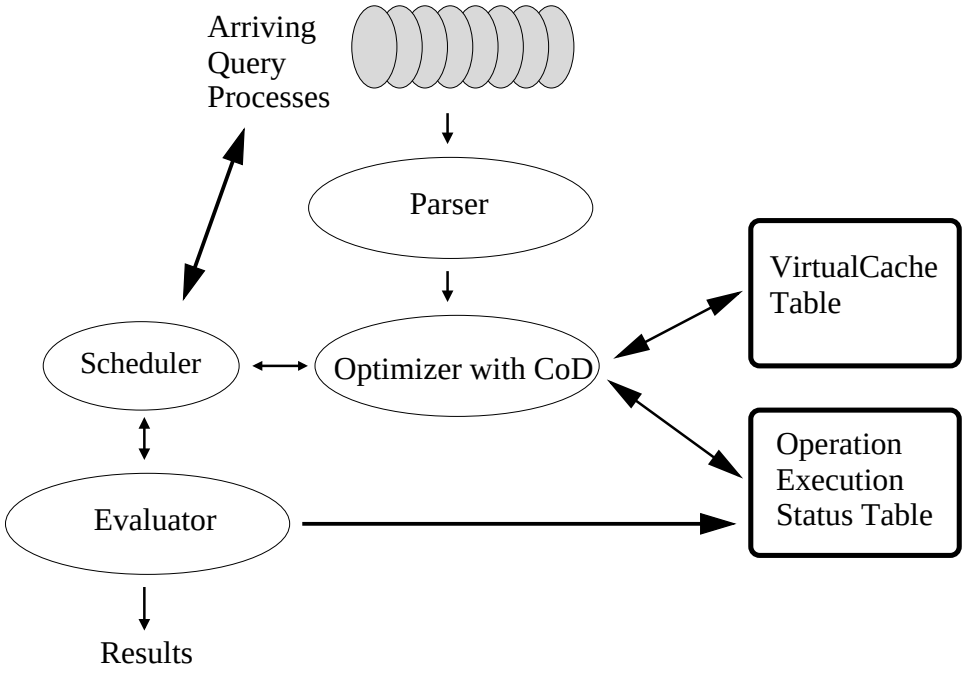


Figure 3.2: System architecture to support Cache-on-Demand.

An arriving query is first compiled by the user process, and optimized to exploit virtual caches. This process requires looking up the VC and OES tables in order to generate an optimal plan. For each virtual cache that can be exploited, we need to know

its operation status, i.e., whether any part of the cache can be salvaged. If the data is already cached or will be cached (when the counter is greater than 0), then we only need to increment the counter, and determine the location of the cache. Otherwise, we check the OES table to determine if the operation is partially evaluated. If it is partially evaluated, then we can indicate that the remaining operations are to be cached. The corresponding plan is also updated to reflect what it has to do for the operation: evaluate the missing portions and obtain the remainder that will be cached by the running queries. The counter of the VC table will also be updated to 1.

Each query is evaluated as a sequence of subqueries (bottom up), where each subquery corresponds to an operation in the plan. To evaluate a subquery, it has to be first submitted to the scheduler which determines whether it is ready for execution. A subquery can be executed only if its input sources are available. Thus, operations that involve base relations are always ready. On the other hand, operations that depend on virtual caches may be blocked until these virtual caches are available. As soon as a subquery is “ready” for execution, the scheduler will mark the subquery. The user process contacts the scheduler to collect ready subqueries and blocks until some subqueries are ready. After finishing execution of these ready subqueries, the user process will update the VC and OES tables (e.g., decrement the counter; if the counter value is 0, then remove the virtual cache information, etc.). This process is repeated until the entire query completes.

3.3 Optimization Strategies

There are essentially two classes of optimization strategies for the Cache-on-Demand framework. The first class comprises two phases. In the first phase, the incoming query will be optimized without considering the currently running queries. In the second phase, the plan obtained from phase 1 is post-processed to exploit any virtual caches that are beneficial. On the other hand, the second class of schemes integrates the two phases into a single phase. In other words, in the one-phase approach, the optimizer broadens its search space to include virtual caches during optimization. While the two-phase methods facilitate reuse of existing optimizer and are expected to have low optimization overhead, the generated plans may not be near optimal. On the other hand, the integrated schemes are expected to produce better plans at a higher optimization overhead and the need to extend or even redesign existing optimizer.

In this section, we shall first present two two-phase strategies, followed by a single-phase scheme. We will then compare these schemes qualitatively, and finally, we will end this section by looking at how the search space of CoD-based schemes can be controlled.

3.3.1 Two-Phase CoD-Based Schemes

As mentioned, in a two-phase optimization strategy, an arriving query is first optimized without considering the currently running queries; and the generated plan is postprocessed to exploit any virtual caches that are beneficial. Here, we shall discuss only the

second phase, since the first phase adopts an existing optimizer to generate a plan. In our study, we used a randomized optimization strategy in the first phase.

Algorithm Conform-CoD

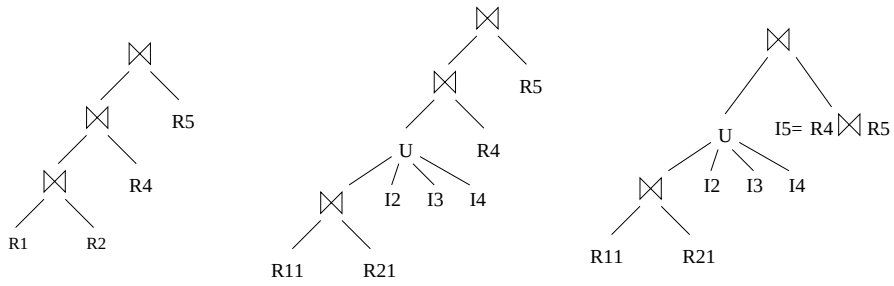
The first algorithm that we consider is called Conform-CoD. Conform-CoD preserves the ordering of the original plan. It simply looks for beneficial virtual caches that can be exploited. Figure 3.3 shows the algorithmic description of Conform-CoD. As shown, it is essentially based on the *breadth-first-search* heuristic. As soon as a node of the plan is found to be beneficial, Conform-CoD revises the plan to take advantage of the virtual cache (lines 4-6) instead of evaluating the subplan rooted by the node. On the other hand, if a node has no beneficial virtual cache (either the cache is not beneficial or there is no matching virtual cache), then its children are further explored (lines 7-10) until the base relations (i.e., the leaf nodes of the query plan that represent the original relations specified in the query) are reached. Figure 3.4 illustrates the algorithm for our running example (in Figure 3.1). Suppose the plan generated for the query Q is a left-deep tree as shown in Figure 3.4(a). From Figure 3.1, we note that only plan P_1 shares the common subexpression $R_1 \bowtie R_2$ with Q. Suppose that $R_1 \bowtie R_2$ is evaluated using a partition join in P_1 as $R_1 \bowtie R_2 = \cup_{j=1}^{4} R_{1j} \bowtie R_{2j}$. Moreover, assume that the answer tuples for $R_{11} \bowtie R_{21}$ have already been produced (without being materialized). Thus, the revised plan for Q under Conform-CoD is as shown in Figure 3.4(b), which indicates that Q has

to reevaluate $R_{11} \bowtie R_{21}$ but it can salvage the join results of the other partitions of R_1 and R_2.

Input: Arriving query's plan, Q
Output: Revised Q that exploits virtual caches

```
1.    enqueue(Queue, Q.root)
2.    while (!empty(Queue)) {
3.          node ← dequeue(Queue)
4.          if beneficial(node) {
5.                revisePlan(Q, node)
6.          } else {
7.                for (i=0; i++; i<number of children of node) {
8.                      if (node.child[i] ≠ base relation)
9.                            enqueue(Queue, node.child[i])
10.               }
11.         }
12.   }
```

Figure 3.3: Algorithm Conform-CoD.



(a) Query plan for arriving query Q (b) Optimized plan by Conform-CoD (c) Optimized plan by Scramble-CoD

Figure 3.4: Illustration of two-phase CoD-based strategies.

Algorithm Scramble-CoD

Because it preserves the ordering in the initial plan, Conform-CoD's effectiveness is limited as it is unable to exploit some virtual caches. For example, for the sample query Q and the two running query plans (in Figure 3.1), Conform-CoD fails to reuse the virtual cache V_3 of plan P_2. To explore the benefits of these virtual caches, we propose a second algorithm called Scramble-CoD. Scramble-CoD allows some reordering of the initial plan to reuse certain virtual caches. Figure 3.5 shows the algorithmic description of Scramble-CoD. As shown, Scramble-CoD operates on the output of Conform-CoD (line 1). While this is not necessary, we have done so to minimize changes to the original plan. For relations/joins that cannot be salvaged by Conform-CoD, we construct a query graph, G, for them (lines 2-6). G is essentially the original query graph minus the relations that can be salvaged and their corresponding edges. Using the same example in Figure 3.4(a), G will contain only two nodes and involves the join of R_4 and R_5. Next, we pick a pair of joins from G that is most beneficial in exploiting the virtual cache (line 10). The plan is then updated to reflect the exploitation of the virtual cache (lines 11-12). If the resultant cost is cheaper, we repeat the process of picking another join (lines 9-18). Otherwise, we consider the current plan optimal and terminate. This algorithm is basically greedy in nature. Figure 3.4(c) shows the result of the optimized plan under Scramble-CoD (assuming that it is beneficial to reuse $R_4 \bowtie R_5$, and the join can be completely salvaged).

Input: Arriving query's plan, Q
Output: Revised Q that exploits virtual caches

```
1.    Q' ← Conform-CoD(Q)
2.    Let m denote the number of base relations in Q' that
            cannot be salvaged by Conform-CoD
3.    if m > 1 {
4.          Let these relations be S_1, S_2, ..., S_m
5.          construct(G)
6.    }
7.    minCost ← cost(Q')
8.    gain ← true
9.    while (gain) {
10.         e ← mostGain(G)
11.         updateGraph(G)
12.         updateQueryPlan(Q')
13.         if cost(Q') < minCost {
14.               minCost ← cost(Q')
15.         else
16.               gain = false
17.         }
18.   }
```

Figure 3.5: Algorithm Scramble-CoD.

3.3.2 Single Phase CoD-Based Scheme: Algorithm Integrated-CoD

As mentioned, single phase schemes attempt to generate an optimal plan by considering the virtual caches during optimization. In this section, we propose the Integrated-CoD scheme for the Cache-On-Demand framework. Figure 3.6 shows the algorithmic description of the proposed scheme. The scheme is based on a randomized strategy, and comprises two loops. The inner loop (lines 4-8) generates a plan, and the outer loop (lines 2-12) determines the number of plans that the algorithm will examine. At each

iteration, the best plan is retained (lines 9-11). Thus, when the algorithm terminates, the final plan, which is near-optimal, is returned.

The algorithm is highly abstracted for simplicity. To generate a plan, the following is done. Let G be a join graph for the incoming query. First, a join involving 2 relations from G or a beneficial virtual cache involving 2 or more relations in G is *randomly* [5] picked (using the procedure **selectRel**(G, VC) in line 5). Next, G is "collapsed" (updated) to reflect that the set of operations in P have been evaluated (procedure **updateProfile(**G**)** in line 6). The partial plan Q is then updated to indicate the sequence of operations (procedure **buildPlan(**Q, P**)** in line 7). This process is repeated until one relation is left in G which corresponds to the final result.

One other issue to consider is the stopping criterion. As in all randomized optimization strategies [44], one simple way is to predetermine the number of iterations at start. In our work, we have set it to 100, i.e., we examined 100 plans in total. Clearly, increasing the number of iterations will increase the optimization overhead with the likelihood of finding the optimal plan. On the other hand, keeping the number of iterations low reduces the optimization overhead but may sacrifice the quality of the plans.

Figure 3.7 illustrates three iterations of the algorithm using our running example (in Figure 3.1), and the different types of plan that may be generated. In iteration 1, we assume that the join between R_1 and R_2 is first selected, followed by the join between

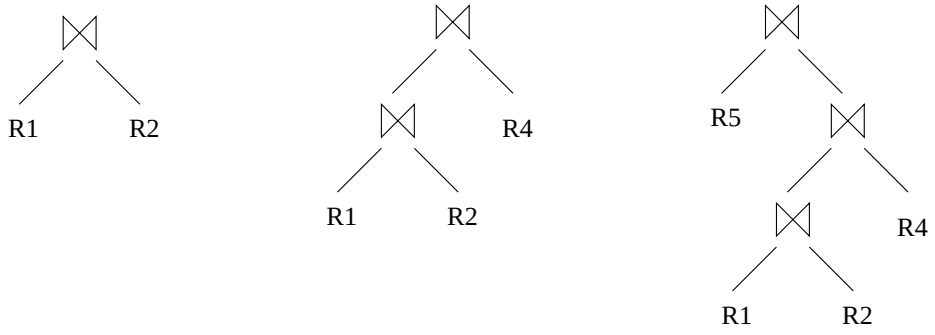
[5] As noted, we have adopted a random strategy. The algorithm also works with any other heuristics that selects relations to be joined.

Input: Join graph of query, G
Output: Optimized plan, P_{opt}, that exploits virtual caches

```
1.    P_opt.Cost ← ∞
2.    repeat
3.        Q ← ∅
4.        repeat
5.            P ← selectRel(G, VC)
6.            G ← updateProfile(G, P)
7.            Q ← buildPlan(Q, P)
8.        until G has only one relation
9.        if Q.Cost < P_opt.Cost {
10.           P_opt ← Q
11.       }
12.   until some terminating criterion is met
13.   return P_opt
```

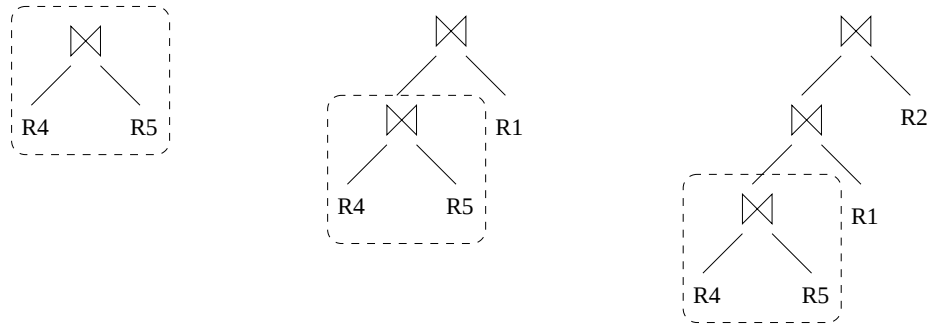
Figure 3.6: Algorithm Integrated-CoD.

($R_1 \bowtie R_2$) and R_4, and finally, the last join is selected. The results of the partial plans are shown in Figure 3.7(a). We note that in this iteration, none of the virtual caches are picked. In iteration 2, suppose the virtual cache involving $R_4 \bowtie R_5$ is picked first (denoted by a dotted bounding box), followed by the join between ($R_4 \bowtie R_5$) and R_1, and finally, the last join is picked. The sequence of partial plans is shown in Figure 3.7(b). In this iteration, we have both virtual cache and joins from G being selected. Finally, in iteration 3, again, we assume that the virtual cache involving $R_4 \bowtie R_5$, followed by the join between relations (R_1 and R_2), and finally, the join between ($R_4 \bowtie R_5$) and ($R_1 \bowtie R_2$) are picked in the given order (see Figure 3.7(c) for the sequence of partial plans).



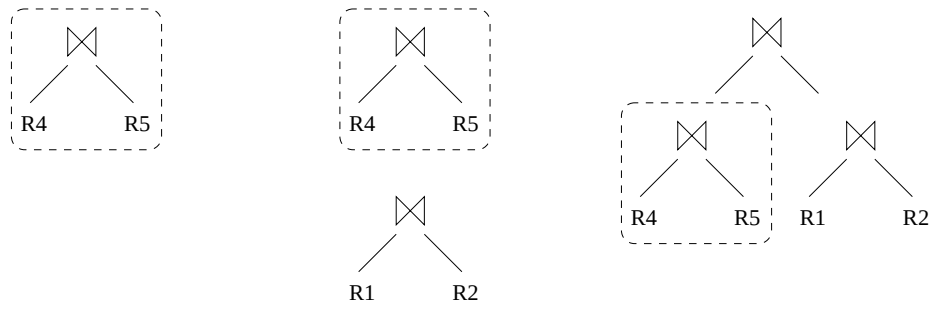
(i) (ii) (iii)

(a) Iteration 1



(i) (ii) (iii)

(b) Iteration 2



(i) (ii) (iii)

(c) Iteration 3

Figure 3.7: Illustration of Integrated-CoD.

3.3.3 A Comparison of the Algorithms

In this section, we shall give a qualitative comparison of the proposed algorithms. Table 3.1 summarizes the comparison.

	Conform-CoD	Scramble-CoD	Integrated-CoD
Performance of Arriving Query	Good	Better	Best
Performance of Overall System	Good	Better	Best
Optimization Overhead	Low	Moderate	High

Table 3.1: Qualitative Comparison of the CoD-Based Schemes.

We note that given a single query, Scramble-CoD can generate no worse plan than Conform-CoD. While it may appear that Scramble-CoD is expected to be superior over Conform-CoD, this is not necessarily the case in terms of the overall system performance (i.e., over a series of queries). In fact, the performance of the algorithms depend very much on the makeup of the workload. Consider the running example in Figure 3.1, and the corresponding plans generated by the two algorithms for the incoming query Q in Figure 3.4. For the query in Figure 3.4, we expect the plan produced by Scramble-CoD to be better. However, consider a query that newly arrived after Q has been submitted for evaluation. Let this newly arrived query be $Q' = R_1 \bowtie R_2 \bowtie R_4$. Now, clearly, had the plan for Conform-CoD been selected for Q, Q' can potentially exploit the intermediate result of Q and need not be evaluated at all! On the other hand, with the plan of Scramble-CoD being adopted, Q' can only salvage $R_1 \bowtie R_2$. Thus, depending on the

net gain in each case, both schemes can outperform one another. However, as shown in our performance study, it is generally the case that Scramble-CoD can lead to better overall system performance.

We also note that Integrated-CoD can generate all possible kinds of combinations, and thus, given sufficient time it can potentially find the optimal plan. On the other hand, we note that the two-phase strategies are quite restricted in their search spaces. For example, in our illustrations, they can never generate the plans produced in Figure 3.7(b). Thus, we can expect Integrated-CoD to generate more superior plans than Conform-CoD and Scramble-CoD.

However, it is also clear that the optimization overhead of Conform-CoD is lower than that of Scramble-CoD. Also, the overhead of Integrated-CoD is expected to be the highest.

3.3.4 Controlling the Search Space

Having looked at the three CoD-based strategies, we note that all the schemes maintain all the virtual caches. For our running example, we have to maintain virtual cache for $R_1 \bowtie R_2$, $(R_1 \bowtie R_2) \bowtie R_3$, $R_4 \bowtie R_5$, and $(R_4 \bowtie R_5) \bowtie R_3$. As the number of relations increases, the number of virtual caches to be maintained can be large and the overhead also increases. We also observe that virtual caches that correspond to a small number of relations are the ones that are more frequently reused, than virtual caches

that involve a large number of relations. As such, we can control the search space by restricting the size of the virtual cache to K, i.e., only virtual cache involving up to K relations need to be maintained. While this may miss some virtual caches (those involving more than K relations), we expect the reduction in the overhead to be worth while. As shall be shown in our experimental results, this is indeed the case. Finding an optimal K value to use is clearly application dependent.

3.4 Experiments

To quantitatively study the performance benefits of CoD, we conducted a series of experiments. We implemented the three proposed strategies, Conform-CoD, Scramble-CoD and Integrated-CoD. We also employed two other schemes as references. The first is scheme NoCache where there is no caching, i.e., each query is executed as if nothing can be salvaged. This allows us to study the benefits of caching.

The second is a speculative strategy, denoted SpCache. SpCache is similar to the Reference-Counting scheme [51], and works as follows: It keeps track of the number of times an intermediate result is (re)used. As queries arrive and depart (completed), the counts are updated accordingly. Given a fixed amount of storage space, we retain as many of the intermediate results as possible beginning from the one with the largest count. A single-phase optimization strategy is adopted to generate query plans that exploit the cached data. In our experiment, we assume 1 GB of storage available. This

value is much larger than that required by the CoD-based strategies (which is no more than 100 MB).

Further, for CoD-based strategies, we investigated the benefits of pruning the search space by limiting the *size* (in terms of the number of relations) of the subquery to be cached. The system tested ran on an 296MHz Ultra SparcII at minimum load.

3.4.1 Experimental Setup

Table 3.2 shows the main parameters used and their default settings. We assumed a closed system, i.e., the number of terminals (and hence users) in the system is fixed. The number of users is given by MPL. Whenever a user's query completes processing, the user will think for $ThinkTime$ sec before submitting the next query. Thus, there will be *at most* MPL queries running in the system. The server has M amount of main memory to be shared by all users, and each page is assumed to be P bytes. For simplicity, we assume that each query will be allocated M/MPL amount of memory.

Our database has D relations, the cardinality of each of which is uniformly distributed over $[C_l, C_h]$ records. Each record is S bytes long. To model the scenario that most of the accesses to the database are directed at a subset of relations, we organize the relations into two groups: FrequentAccessed and SeldomAccessed. The former represents relations that are frequently accessed, while relations in the latter group are less frequently demanded. We have assigned α% of the relations to the group FrequentAc-

Notation	Meaning	Default Values
	System Parameters	
MPL	number of users	50 (5-80)
$ThinkTime$	mean think time of users (exponential distribution)	10 sec
P	page size	4KB
M	system memory	128 MB
	Database Parameters	
D	number of relations in the database	40
$[C_l, C_h]$	cardinality of each relation (uniform distribution)	[100K,200K]
S	size of a record	200 bytes
α	% of relations belonging to FrequentAccessed group	20%
	Query Parameters	
Q	number of basic queries	4 (8, 12, 16, 20)
N	number of relations per query	8 (4, 6, 8, 10, 12, 14)
β	probability of a relation in a query belonging to the FrequentAccessed group	0.8
γ	probability of a query belonging to the basic queries	0.8
δ	probability of a relation in a query to be replaced	0.2
K	maximum number of relations in a virtual cache	N (2,4,8)

Table 3.2: Cache-On-Demand's Experimental Parameters.

cessed and the rest of the relations are in the group SeldomAccessed.

To generate the queries, we adopt the following approach.

- First, we create a complete graph, $\mathcal{G}$, on all the relations, where an edge between two relations represents a join predicate between them. The edge is labeled by a value between 0 and 1 to denote the join selectivity. The join selectivity is determined such that the join result size is also in the range $[C_l, C_h]$ records.

- Second, we pick Q connected acyclic subgraph of N relations from $\mathcal{G}$ to be the *basic* queries. The subgraph (or query) comprising N relations is obtained by the following steps: (a) We determine the N relations using the following procedure: We generate a random number between 0 and 1; if the value is smaller than β (in probability), then we pick a relation randomly from the FrequentAccessed set, else we pick a relation randomly from the SeldomAccessed set. This allows us to model sharing of relations using the rule that $(\beta \times 100)$% of the relations in a query belong to the FrequentAccessed group. (b) For the kth ($k > 1$) relation picked, we add an edge between the kth relation and one of the $k - 1$ relations already picked (the relation is determined randomly). The selectivity information are obtained from the corresponding edges in $\mathcal{G}$.

- Third, to model variations of queries, the actual query submitted by a user is given by the following procedures: (a) The user picks one of the basic queries randomly. Let this query be q. (b) A random number between 0 and 1 is generated. If its value is smaller than γ, then the user submits q. Otherwise, the user will generate another random number between 0 and 1. Let this number be δ. Then $\delta * N$ relations in q of degree 1 will be randomly picked and removed with its corresponding edge. Let us denote the resultant subgraph as q'. Another $\delta * N$ relations from amongst the database relations will then be picked and added into q' in the same manner as discussed in the second step. The new q' is the query submitted by the user.

The basic queries can be seen as partitioning the users across different applications. By allowing the basic queries to be varied, we can model differences in queries amongst users of the same applications. Moreover, the procedure of generating queries allows us to study the degree of overlap in queries. On one extreme, if we set Q and γ to 1, then we have an application where all users share the same query. On the other extreme, by setting Q to a large value, γ to 0 and δ to 1, it is most likely that all users will submit different queries and not much can be expected to be salvaged.

In our experiments, we used the parameter K to control the maximum number of relations that can participate in a virtual cache. Recall that a higher value of K relaxes the number of relations allowed for a subquery for it to be cached, and a smaller value restricts it. For example, if $K = N$ (i.e., maximum value that K can take), we can potentially cache the intermediate results that involve all the relations in the query. If $K = 4$, then we only need to maintain the cache that involves no more than 4 relations. In this way, the VC tables are bigger for the formal but smaller for the latter.

For each query, we optimized it using a randomized algorithm [44]. For NoCache, the plan is evaluated as it is. For Conform-CoD and Scramble-CoD, the plan is post-processed to exploit caching under the respective schemes. For SpCache and Integrated-CoD, the same randomized algorithm is used; however, the search space is expanded to include the caches.

For each experiment, we run a total of 10200 queries. The first 200 queries are used to “warm-up” the system. For each of the remaining queries, we record its elapsed time

(the time from initiation to the time that all answers are produced). We note that the time to write the final result of each query to disk is not included in the elapsed time calculations. We felt that this was more realistic because aggregates are often used on large join queries to condense the final result into a report. Moreover, the time to write the final result to disk would be the same for each scheme.

Since queries may vary wildly in terms of elapsed time, results are scaled by the elapsed time for NoCache. For example, let $Conform_i$ and $NoCache_i$ denote the time to execute the plan generated by Conform-CoD and NoCache for query i, respectively. Then, the scaled average for Conform-CoD for n queries is calculated as

$$\frac{1}{n}\sum_{i=1}^{n}\frac{Conform_i}{NoCache_i}$$

We shall denote the scaled average for Conform-CoD, Scramble-CoD, Integrated-CoD and SpCache as Conform, Scramble, Integrated and SpCache, respectively.

3.4.2 Experiment 1: Effect of MPL, Number of Users

In this experiment, we vary MPL from 5 to 80. The result is shown in Figure 3.8. First, we observe that cache-based algorithms (SpCache and CoD-based schemes) performed significantly better than the algorithm that used no caching. From the figure, we see that SpCache is more than 20% faster than NoCache. The gain of the CoD-based schemes over NoCache is much more significant - Conform-CoD, Scramble-CoD and Integrated-CoD requires, on average, no more than 80%, 55% and 50% of the response time for

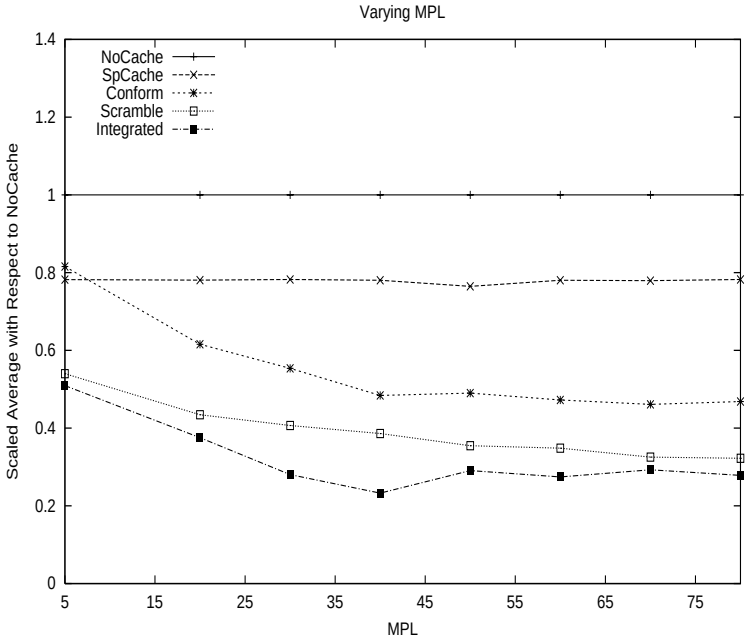


Figure 3.8: Varying MPL.

NoCache. In fact, the response time for the three methods can be reduced to as low as 50%, 40% and 20% that of the NoCache scheme respectively. This clearly demonstrated the effectiveness of CoD-based strategies in exploiting running queries to provide cached data for incoming queries.

Second, we observe that SpCache is inferior to the CoD-based schemes. In fact, the CoD-based strategies can be as much as 30%-60% faster than SpCache. This can be attributed to the fact that SpCache is unable to salvage answers of running queries that are not cached. This result also shows the significant gain that can be derived from careful design of caching strategies: even an integrated scheme where the optimization process considers cached data (e.g., SpCache) may still perform worse than two-phase

approaches that exploit dynamic caching of intermediate results (we will defer the explanation on the exception between SpCache and Conform-CoD for small MPL to a later section).

Third, we note that the performance of the three CoD-based algorithms typically improved with increasing MPL values. This trend is such because as MPL increases, the search space of the virtual space increases, leading to a higher probability of getting a match and reuse option. Moreover, as MPL increases, more queries share the same basic queries, and thus a materialized virtual cache benefits more queries. On the other hand, we see that MPL has little effect on the relative performance between SpCache and NoCache. This is because the set of frequently used caches are always available, while those that are less frequently used cannot be salvaged. In addition, the size of the cache (1 GB) is sufficient to cache a substantial amount of intermediate/final answers.

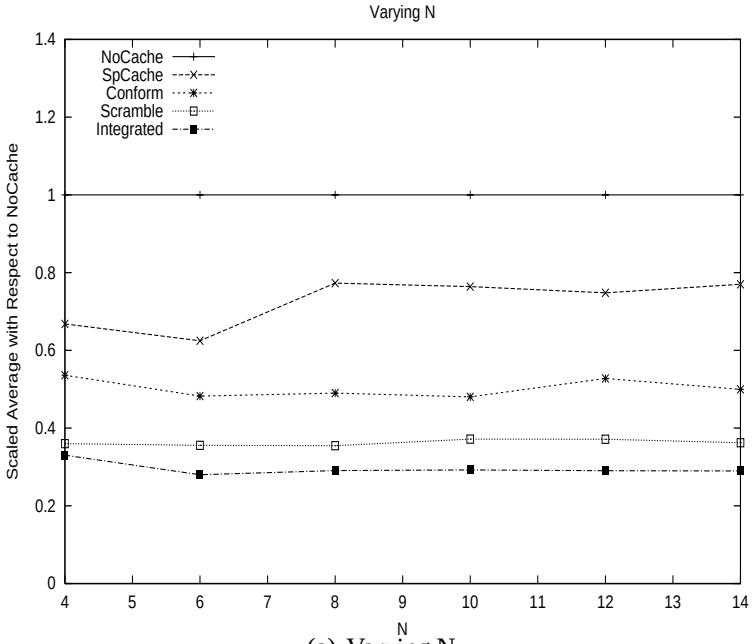
Finally, comparing Conform-CoD and Scramble-CoD, we note that Scramble-CoD is slightly superior over Conform-CoD. The improvement comes from a greater opportunity to exploit more virtual caches. We also observe that Integrated-CoD performs best. The reason for this better performance is attributed to the improved optimized plans that are being produced by considering the virtual caches while optimizing the plans. This improvement is as expected since the awareness of caching/salvaging happens earlier as part of the optimization process rather than as a post-processing step. As compared to the Scramble-CoD approach, this early awareness provides a higher chance of exploiting the maximum benefits possible where changes to the initial plan are still in process. Fur-

ther, Scramble-CoD makes progressive changes to the plan which is afterall limited only to a local optimum. Thus, we see that an integrated approach is superior over two-phase approaches.

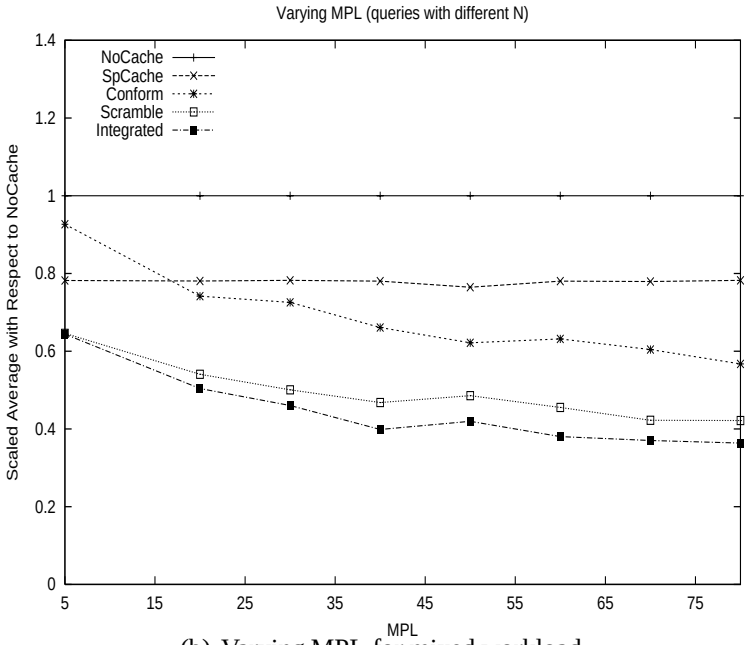
3.4.3 Experiment 2: Effect of N, Number of Relations in a Query

In this experiment, we study the effect of N, the number of relations per query, by varying N from 4 to 14. Looking at the result shown in Figure 3.9(a), we see that the CoD-based strategies remain superior over the NoCache scheme. The relative performance of the algorithms also remain the same, with Integrated-CoD performing the best, and Scramble-CoD outperforming Conform-CoD, and the CoD-based schemes performing better than SpCache. There are, however, two interesting findings. First, we observe an undulating trend for the CoD-based algorithms. The cause of such effect is due to the unpredictable amount of sharing of common subplans between queries as more combinations of generated plans are allowed with the relaxed number of relations in a query. Second, we also note that SpCache's gain over NoCache is more significant for small number of relations. This is because for small number of relations, there are fewer combinations of intermediate results; thus, most of the cached content can be salvaged.

We also experimented with a mixed workload to study the effect when different queries involve different number of relations. In this experiment, queries are randomly assigned 4 to 8 relations. The result of the experiment is shown in Figure 3.9(b) as MPL



(a) Varying N



(b) Varying MPL for mixed workload

Figure 3.9: Effect of N.

varies from 5 to 80. We observe that the performance follows the trend of that described for Figure 3.8, except that the overall performance gain of CoD-based schemes over NoCache is lower. This can be explained by the fact that the search space of the Virtual Cache is reduced when the set of running queries comprises many queries of small N, resulting in a smaller gain.

We also noticed that Conform-CoD performs worse than SpCache for small MPLs. There are two reasons for this. First, Conform-CoD is a two-phase approach. Second, the opportunity for salvaging virtual caches is lesser when the MPL is small. Having a mixed workload with queries that comprise small number of relations further reduces the search space of the virtual cache.

3.4.4 Experiment 3: Effect of Degree of Overlap

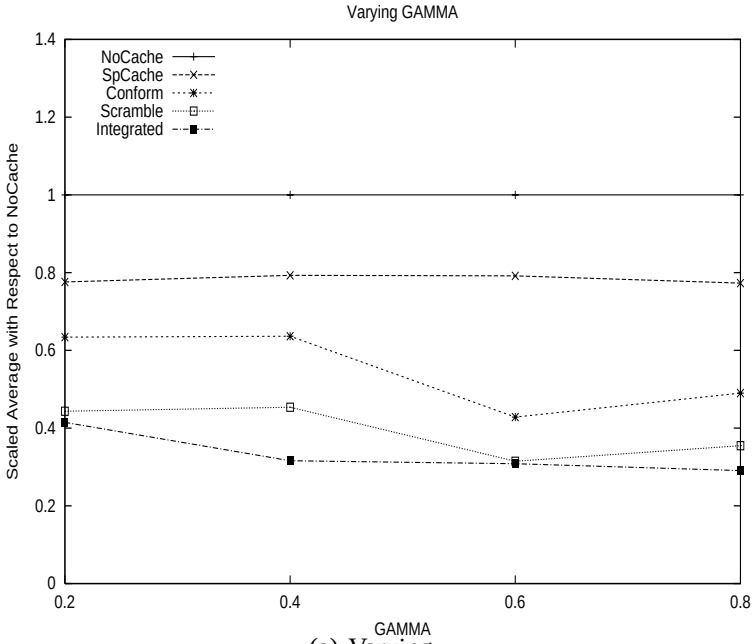
In this experiment, we study the effect of the degree of overlap. This can be controlled by three parameters: γ, δ and Q. We first study the effect of varying γ from 0.2 to 0.8. By increasing γ, the probability of an incoming query belonging to the same basic queries increases. Hence, the amount of sharing of similar subplans between queries increases, leading to an improvement in performance by the CoD-based schemes, as seen in Figure 3.10(a). We note that SpCache is not much affected by γ. This is because of the large cache size (1 GB) that was used.

We next study the effect of δ, which we vary from 0.2 to 0.8. Unlike γ, as δ increases,

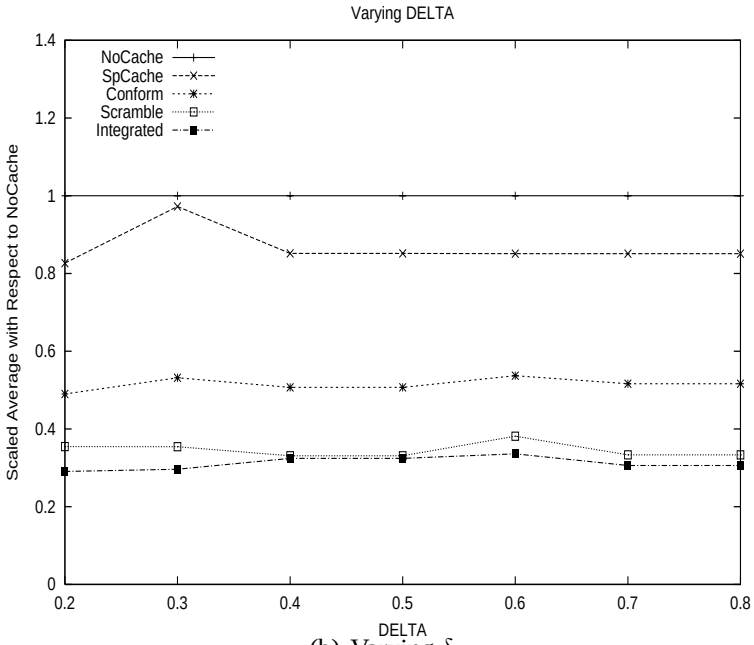
the percentage of relations in an incoming query to be replaced increases. Hence, differences between queries increase as well, leading to less sharing of similar subplans. This effect is confirmed in our results as shown in Figure 3.10(b). For SpCache, the poorer performance when $\delta = 0.3$ is due to the effect of the randomization in picking the relations, and should be seen as an exception.

Recall that in Experiment 1, we have used the default setting of $\gamma = 0.8$ and $\delta = 0.2$. We also conducted an experiment to see how the proposed schemes perform when $\gamma = 0.2$ and $\delta = 0.8$. This represents a fairly pessimistic scenario where the degree of overlap is not very high. The result, as we vary the MPL from 5 to 80, is shown in Figure 3.11(a). As expected, because of the fewer number of common subplans, the gain of the CoD-based schemes over NoCache is less (compared to the result in Experiment 1). However, the CoD-based schemes remain clearly effective. It is also interesting to observe that both Integrated and Scramble-CoD outperform Conform-CoD by a wider margin. This is because, with fewer common subplans, Integrated-CoD and Scramble-CoD have a greater opportunity to exploit common subplans than Conform-CoD. Moreover, the effect of a newly arrived query having the same basic query as earlier queries diminishes. On the contrary, we observe that SpCache's performance is not very much affected. As noted before, this is because of the large cache size that we have used that allows it to keep a fairly substantial number of intermediate/final results.

Finally, we also study the degree of overlap by looking at the effect of Q, by varying Q from 4 to 20. In Figure 3.11(b), we see that as Q increases, the performance of all

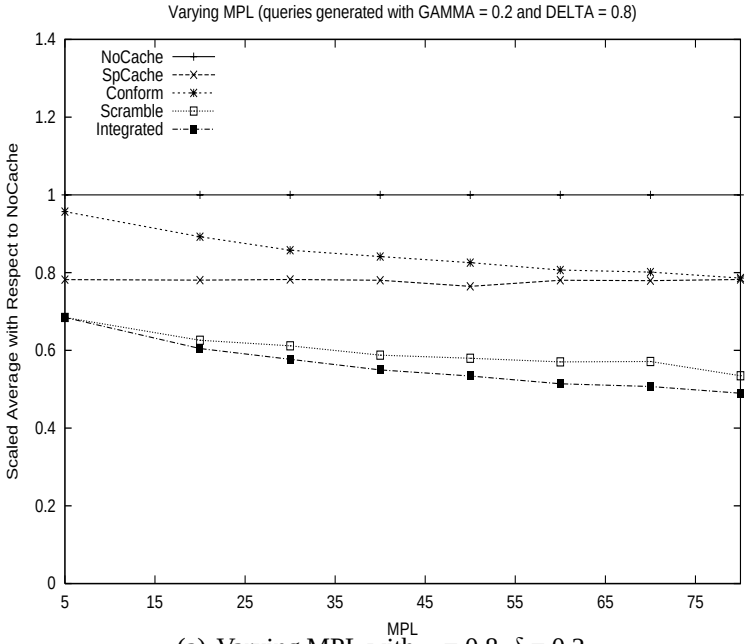


(a) Varying γ

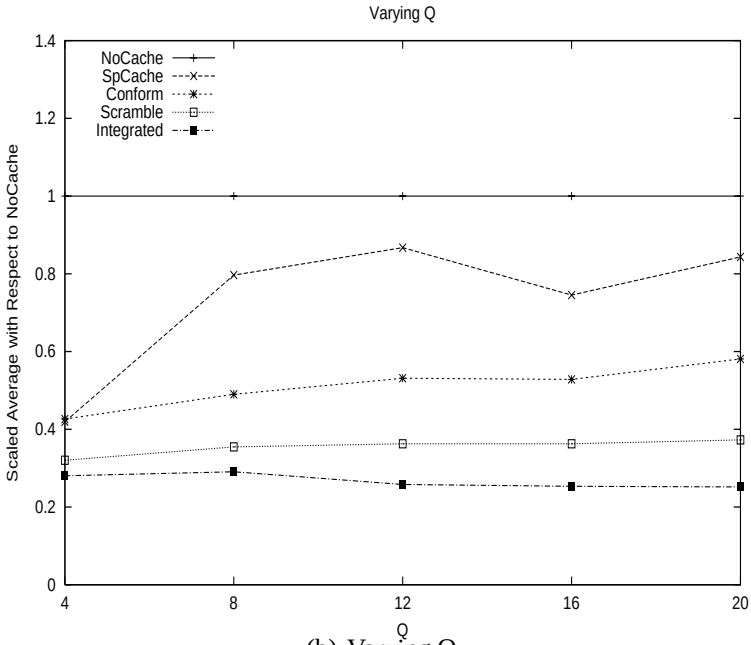


(b) Varying δ

Figure 3.10: Effect of γ & δ.



(a) Varying MPL with $\gamma = 0.8$, $\delta = 0.2$



(b) Varying Q

Figure 3.11: Effect of MPL & Q.

cache-based schemes dips. This is so because as Q increases, the probability of choosing the same basic query drops, leading to a higher degree of difference between queries.

3.4.5 Experiment 4: CoD Schemes with Controlled Search Space

In the previous experiments, we have set K to be the maximum possible value, i.e., the number of relations. In this experiment, we would like to study the effect of controlling the search space by setting K to 2, 4 and N. We perform these experiments for all the three CoD-based schemes. The results for Conform-CoD, Scramble-CoD and Integrated-CoD are shown in Figure 3.12, 3.13 and 3.14 respectively.

From Figure 3.12(a), we see that Conform-CoD with $K = N$ performed the best. This is expected as a larger search space is available since all N relations are used for caching. On the other hand, the optimization overhead of Conform-CoD for $K = N$ is higher, as it searches a larger VC table, and examines more virtual caches. Nevertheless, the improvement is about 20% which is a significant performance increase for Conform-CoD using $K = N$ as compared to Conform-CoD using $K = 4$. Hence, Conform-CoD using $K = N$ is preferred. Interestingly, as we make a switch between the γ and the δ values (effectively reducing the number of common subqueries in the general pool), we observe a slightly different picture. As shown in Figure 3.12(b), Conform-CoD using $K = N$ and $K = 4$ perform equally well. The greater decrease in performance of Conform-CoD using larger K value is due to the drastic drop in the number of common

subqueries which affects the larger subqueries more than the smaller subqueries. Hence, in general, when there is nothing much to salvage, pruning away the larger common subqueries becomes favorable.

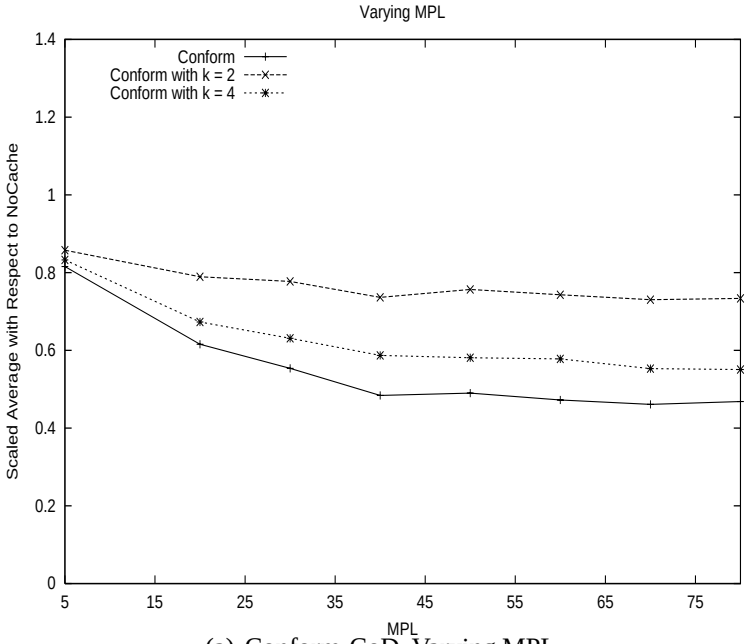
The results for Scramble-CoD (see Figure 3.13(a) and (b)) show similar trend. However, we note that a K value of 4 is now sufficient to produce good performance when the data sharing is high (Figure 3.13(a)), and a K value of 2 is good enough when the data sharing is low (Figure 3.13(b)). This is because Scramble-CoD searches a larger space.

Finally, as shown in Figure 3.14 (a)&(b), Integrated-CoD also shows similar trend as Scramble-CoD – a K value of 4 suffices to produce good quality plans for high data sharing, and a value of 2 will do for low data sharing applications.

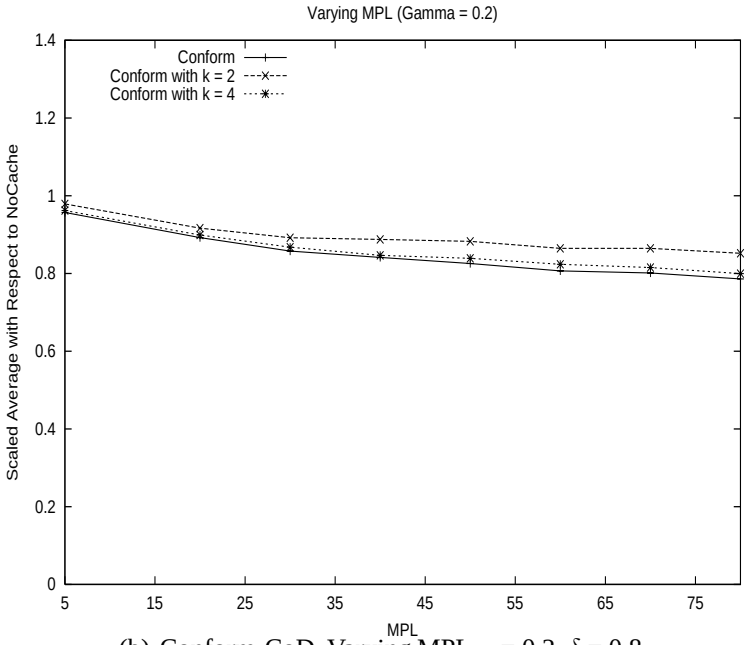
3.4.6 On Optimization and Processing Overhead

So far, we have seen that among the CoD-based strategies, Integrated-CoD offers the best performance, followed by Scramble-CoD and finally Conform-CoD. Here, we shall look at the optimization overhead incurred by each scheme. Figure 3.15 & 3.16 shows the overhead incurred as the number of relations varies from 4 to 14. These results are obtained based on the average running times for each experiment over 200 runs.

In Figure 3.15(a), we compare the optimization overhead of the various schemes. As shown, Conform-CoD requires the least amount of optimization overhead, while

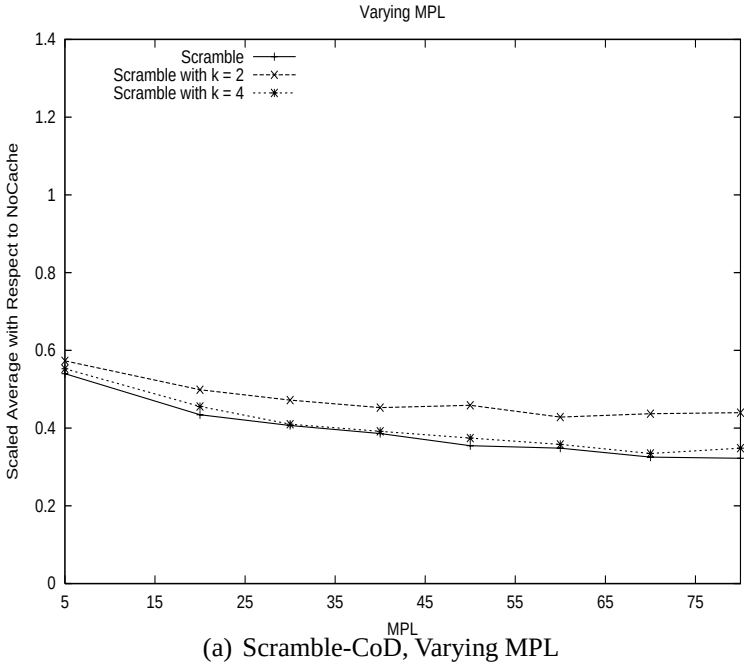


(a) Conform-CoD, Varying MPL

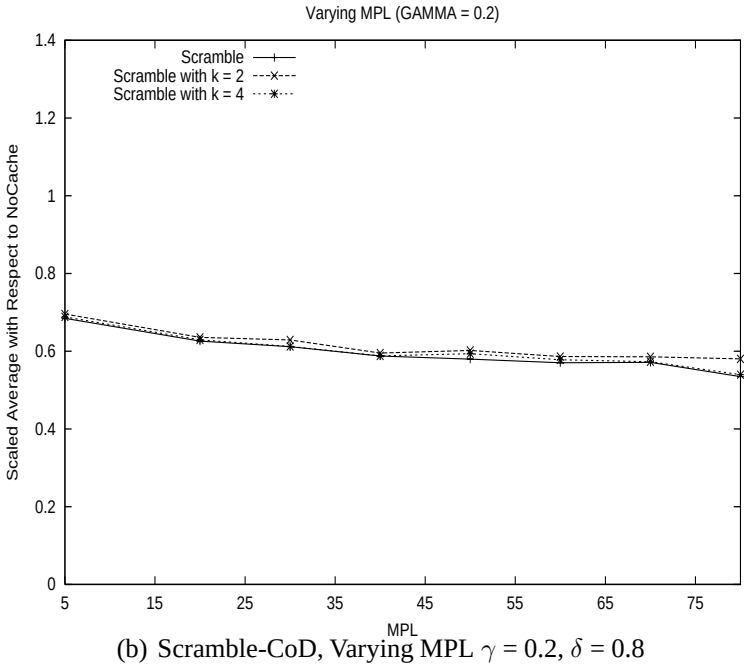


(b) Conform-CoD, Varying MPL γ = 0.2, δ = 0.8

Figure 3.12: Conform-CoD.

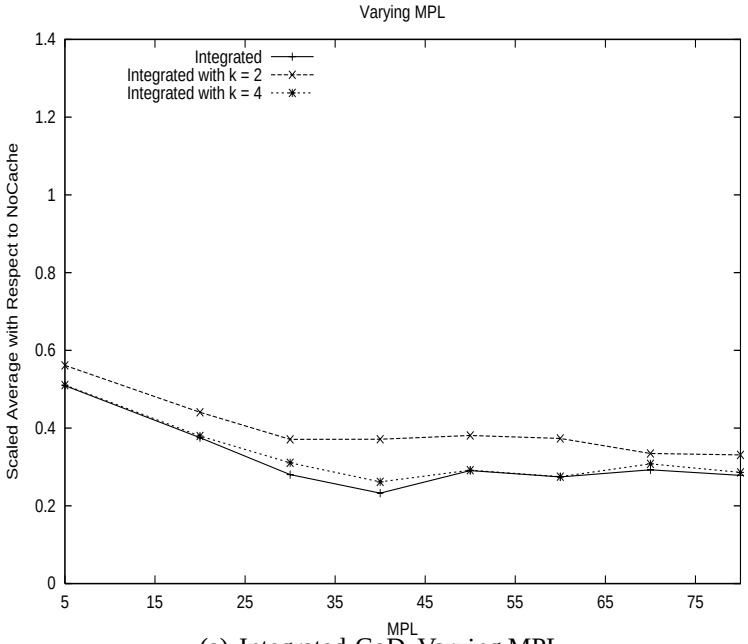


(a) Scramble-CoD, Varying MPL

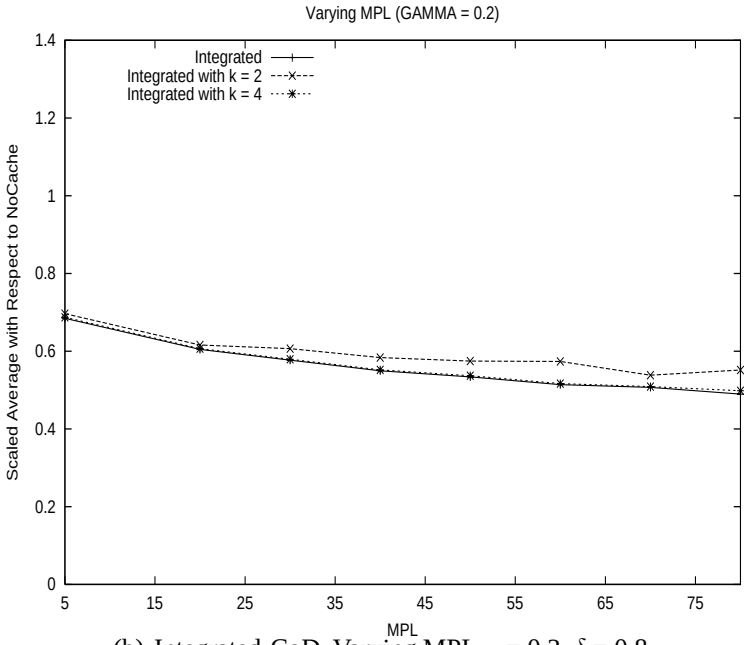


(b) Scramble-CoD, Varying MPL $\gamma = 0.2$, $\delta = 0.8$

Figure 3.13: Scramble-CoD.



(a) Integrated-CoD, Varying MPL



(b) Integrated-CoD, Varying MPL $\gamma = 0.2$, $\delta = 0.8$

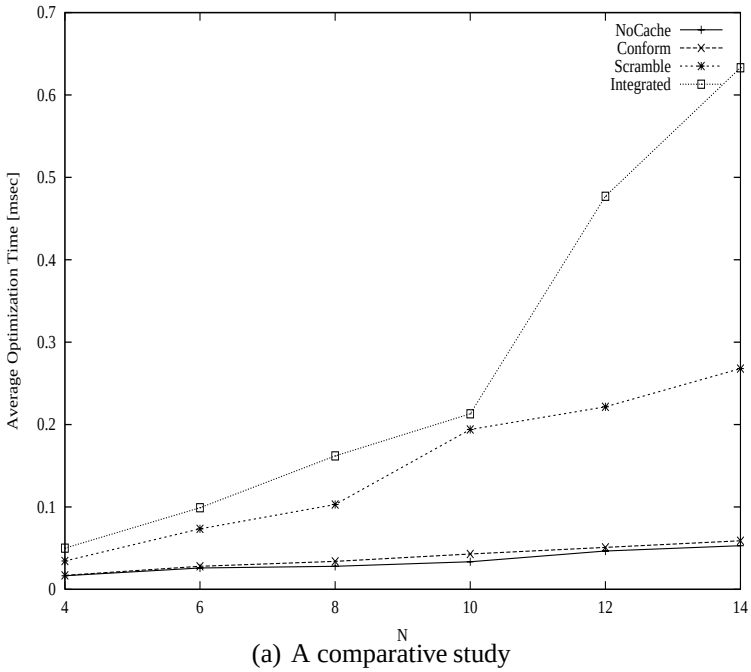
Figure 3.14: Integrated-CoD.

Integrated-CoD consumes the most running time. In fact, Integrated-CoD takes more than 10 times the running time of Conform-CoD, and more than twice the amount of time required for Scramble-CoD. This is expected in view of the larger search space the optimizer has to explore. Scramble-CoD also takes more than five times the overhead of Conform-CoD for the same reason. It is interesting to note that Conform-CoD requires only a marginal increase in overhead compared to NoCache scheme.

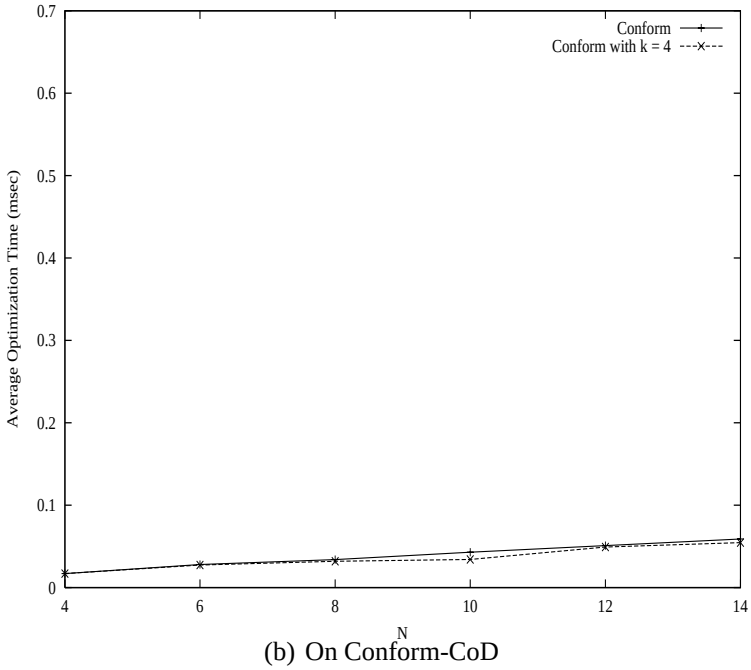
We also note that the all the schemes' optimization time increases with the number of relations. This is expected since more relations call for more time to be spent to find an optimized plan. However, while the optimization overhead for Integrated-CoD increases exponentially, the overhead for the two-phase methods increase almost linearly. This is due to the greedy nature of the algorithms.

Figure 3.15(b) & 3.16 show the effect of fixing the number of relations to be cached on each of the proposed schemes. Here, we just show the case when K is set to 4, i.e., only virtual caches involving at most 4 relations are considered during optimization. For Conform-CoD, since its overhead is not much, the gain in fixing K to a small value is also not significant (see Figure 3.15(b)). However, for both Scramble-CoD and Integrated-CoD, we note that the restriction of K to a small value can cut down on the optimization overhead: for Scramble-CoD, the overhead can be reduced by as much as 35% (see Figure 3.16(a)); for Integrated-CoD, the gain is only about 20% (see Figure 3.16(b)).

We also observe that the gain in (reducing) overhead increases with larger number

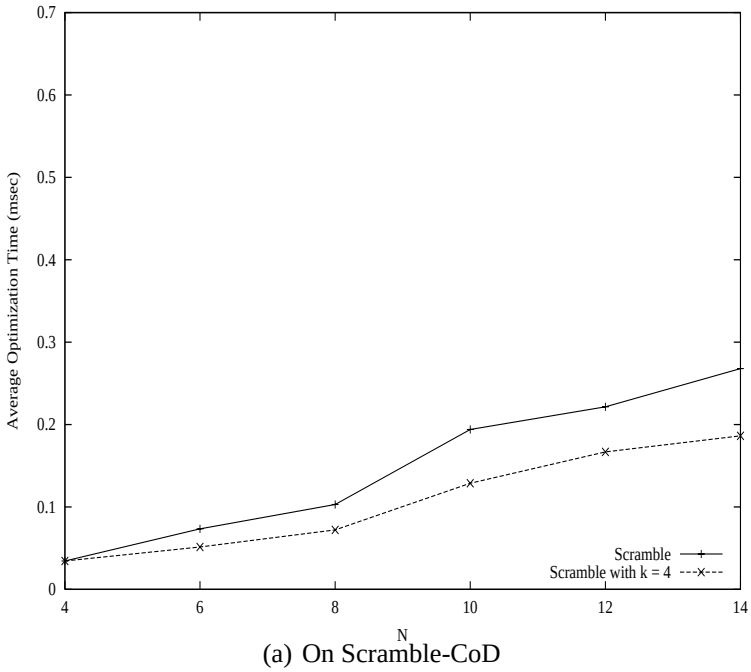


(a) A comparative study

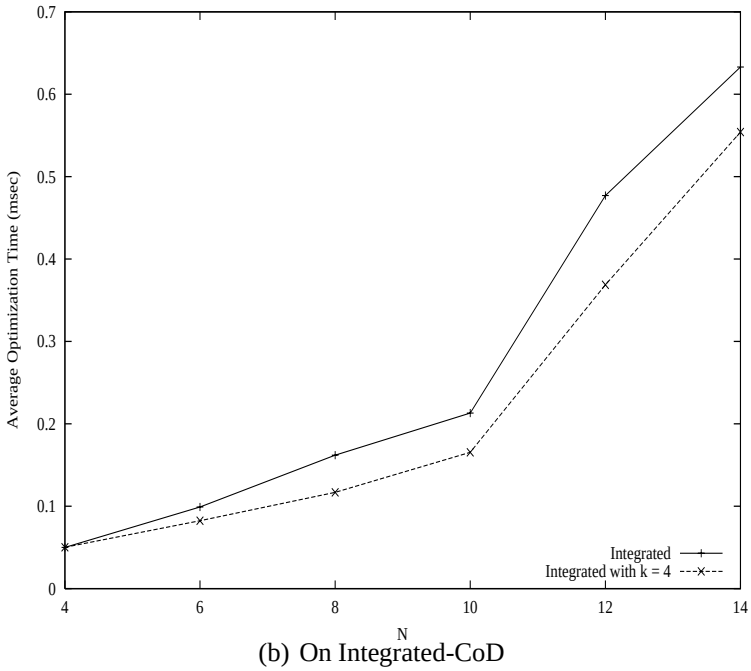


(b) On Conform-CoD

Figure 3.15: Optimization overhead of Combined & Conform-CoD.



(a) On Scramble-CoD



(b) On Integrated-CoD

Figure 3.16: Optimization overhead of Scramble & Integrated-CoD.

of relations. This is reasonable since the search space is more significantly reduced for larger number of relations (than for smaller number of relations).

We have also measured the influence (processing overhead) of informing running queries that is incurred by the CoD-based schemes. On average, this cost turns out to be no more than 1 ms, which is negligible.

3.5 Summary

In this chapter, we have proposed and studied a novel Cache-on-Demand framework. Essentially, CoD allows beneficial intermediate/final answers of existing running queries to be cached to speed up an arriving query's evaluation. We have proposed three caching strategies based on the CoD framework - Conform-CoD and Scramble-CoD are based on a two-phase optimization framework, while Integrated-CoD operates on a single phase optimization.

We conducted extensive performance study to evaluate their performance. Our results showed that CoD-based schemes can provide substantial performance improvement over a scheme that does not support caching. Our results also showed that the proposed schemes outperform a speculative scheme. Moreover, single-phase method is shown to outperform two-phase schemes.

Chapter 4

Cache-On-Demand with Pipelined Plans

In this chapter, we propose the Cache-on-Demand mechanism with pipelined plans and other extensions, which builds on the CoD framework to further improve the overall performance of the query evaluation engine. It integrates three new techniques to realize this performance gain. The first method exploits intraquery parallelism where a sequence of operators within a query execution plan are executed in a pipeline. The second method explores the advantage of keeping multiple plans to increase the match space of CoD virtual caches at the expense of memory and comparison overhead. Lastly, the execution orders of plans may be reordered by the plan scheduler. We also address several issues for these strategies to have a higher chance of success. We implemented the extensions and evaluated its performance. Our results provide insights into the possibilities of using

these strategies to improve the overall performance of a multi-user system.

4.1 The Mechanisms

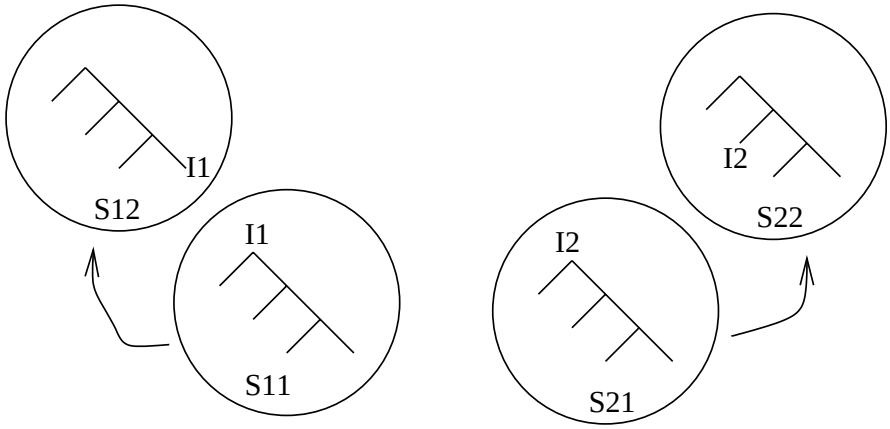
In this section, we shall examine the three mechanisms that extend the query processing capabilities of the CoD framework: (a) salvaging pipelined plans, (b) enlarging the opportunity for sharing by keeping multiple plans, and (c) reordering of execution plans at runtime. Before looking at the proposed techniques, let us look at how pipelined plans are evaluated.

4.1.1 Evaluation of Pipelined Plans

In our previous proposal, the assumption is that relations sizes are typically larger than the main memory. Here, we relax the assumption, and consider an environment where multiple relations can fit in the main memory. This is increasingly becoming possible as today's systems are typically equipped with large main memory. As such, it is possible to exploit the benefits of pipelined plans.

In our extension work, we shall restrict to hash joins only. We also restrict our work to segmented right-deep trees, which are commonly used in multi-processor environments [18, 85]. A segmented right-deep tree comprises a sequence of memory-resident segments, each of which is a right-deep tree. Each segment is evaluated by first constructing the hash tables of all building relations. As each tuple of the probing relation is

retrieved, it is used to probe the building relations for matches, and answers are returned immediately. Figure 4.1 shows two segmented right-deep trees. In plan A, there are two segments, S_{11} and S_{12}. Segment S_{11}'s output is used as the probing relation of S_{12}. In plan B, there are also two segments, S_{21} and S_{22}. However, the result of segment S_{21} is used as a building relation of S_{22}.



(a) Plan A: Segment S11's output is the probing relation in segment S12.

(b) Plan B: Segment S21's output is a building relation in segment S22

Figure 4.1: Examples of segmented right-deep trees.

In the extended CoD, each query is evaluated as a sequence of sub-queries, where each sub-query corresponds to a segment in the plan. Referring to Figure 4.1, for plan A, segment S_{11} will be evaluated before S_{12}. To evaluate a segment, it has to be first submitted to the scheduler which determines whether they are ready for execution. A sub-query is ready for execution only if its input sources are available. Thus, segments that involve base relations only are always ready. On the other hand, segments that

depend on intermediate results (either from other segments of the same query or caches from other queries) may be blocked until these intermediate results are available. As soon as a sub-query is "ready" for execution, the scheduler will mark the sub-query.

In our extension work, we are assuming a single CPU environment. As such, ready segments joined a waiting queue. At any one time, only one segment is evaluated, and upon completion, another ready segment in the queue is evaluated.

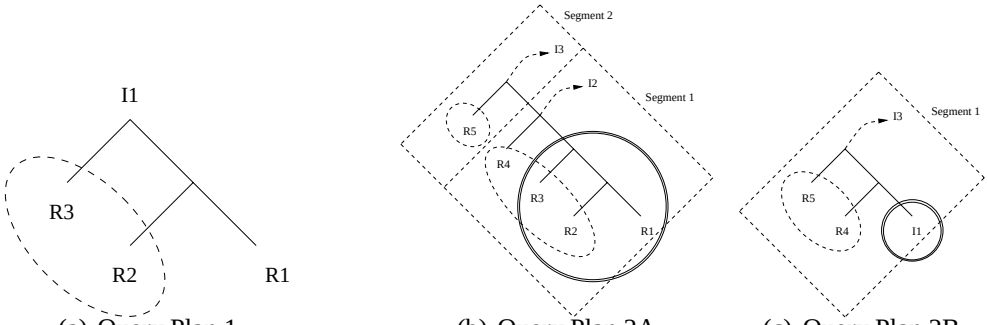
4.1.2 Salvaging Segmented Pipelined Plans

In the extended CoD, an incoming query tries to salvage sub-plans from queries that have arrived earlier and are still in the system. For a pipelined segment, it is best if the entirety of the segment can be salvaged since the segment results are materialized anyway. However, if only part of the segments can be salvaged, the additional overhead is for the intermediate result to be also written out.

To illustrate, suppose we have a query plan as shown in Figure 4.2(a) in the system. Assuming that we have another query, Query 2, as seen in Figure 4.2(b) which enters the system and follows after the initial query in Figure 4.2(a). Here, the plan has 2 segments and requires 2 intermediate results (actually 1 intermediate and 1 final result) to be written out. We observed that this new query has a common subplan that matches exactly that of the entire initial plan. In this case, we could request the **Executor** of the initial query which may not have executed the query to materialize the final result

I1 for use by this query. The resultant plan as a result of exploiting the *I1* is shown in Figure 4.2(c). In this case, it may be possible to fit the entire plan into the memory and hence only require 1 segment and the writing of only 1 intermediate result.

Optionally, Query 2 can disregard salvaging for common subplans and execute the plan on its own. The latter option will be good if the size of the intermediate result *I1* is very much larger than the combined size of all the relations which were used to generate it. However, it uses more memory as building relations will have to be brought in for the join to occur, which accounts for 2 segments.



(a) Query Plan 1 (b) Query Plan 2A (c) Query Plan 2B

Figure 4.2: Variations of Query Plan 2

An estimated I/O cost comparison is as follows ($|R|$ denote the size of R in terms of number of pages):

Cost for reading in Query Plan 2(a) = $(|R1| + |R2| + |R3| + |R4|) + (|I2| + |R5|)$

Cost for writing out Query Plan 2(a) = $|I2| + |I3|$

Cost for reading in Query Plan 2(b) = $|I1| + |R4| + |R5|$

Cost for writing out Query Plan 2(b) = $|I3|$

Hence, if $|I1| - (|R1| + |R2| + |R3|) < 2|I2|$ then **Query Plan 2(b)** will be a better plan compared to **Query Plan 2(a)**, otherwise the latter is a better plan. However, we did not consider the cost of writing out (the overhead of $|I1|$) the initial plan needed by **Query Plan 2(b)**. Nevertheless, this cost will be divided among all the later queries that share the initial plan. So, this cost will be small if many queries shared a common sub-plan.

We note that when **Query Plan 2(b)** has a higher I/O cost compared to **Query Plan 2(a)**, it may still be beneficial to keep the former plan. This is mainly due to the presence of a nondeterministic delay which occurs between the executions of segments. This delay is due to the time needed for the releasing and acquiring time and space resources for each segment execution. Moreover, if each plan were to execute only one segment at a time during its acquired time period, than the segment to follow in the same plan will only be allowed to execute after all the other plans had expired their time period.

4.1.3 Generating Alternative Sub-plans

Under the basic CoD framework, each arriving query looks at the current virtual caches and see how they can be reused. If necessary, the query plan of the arriving query will have to be revised to exploit the cache. It is, however, possible, that the incoming query shares some common relations and operations with the running queries but not in the

orders in the plans. As such, the commonalities cannot be exploited. For example, suppose we have a running query whose plan is as shown in 4.3(a), and an incoming query whose plan is $m1$ as shown in 4.4(a). Now clearly, both queries actually share the join operation $T_1 \bowtie T_3$, but this cannot be exploited since the virtual cache does not reflect this expression.

In order to broaden the search space, we propose the duality problem: instead of asking how an incoming query can be adjusted to exploit running queries, we ask how the query plans of running queries can be adjusted to facilitate further exploitation of virtual caches. Referring to our example again, we can potentially swap the query plan to that of 4.3(b) without affecting the query answers and allows the incoming query to reuse the join result $T_1 \bowtie T_3$. (Of course, the decision to eventually revised the plan or not will have to depend on a cost-benefit analysis).

There are two possible solutions. First, we can determine at runtime what plans to revise that may reap the most gain. This, however, may incur costly runtime overhead. An alternative solution, which we prefer, is to keep certain number, say k, of alternative plans per segment of a query plan. This approach is feasible for the following reasons: (a) the intermediate results of a segment is the same regardless of the ordering of the relations; (b) the number of plans is only a multiple of the number of segments, i.e, each segment keeps k sub-plans for the segment, so the space requirement is not significant; (c) the k plans to be maintained are expected to be the better plans that fit the segment, i.e., the best k plans; this avoids the need to examine expensive plans that may fit the

arriving query but not the running query itself; in other words, we do not want to penalize the current query.

There are three issues that need to be considered. First, we need to decide what are the alternative plans that are worth keeping. Secondly, we need to have a way to keep track of the benefits for materializing each virtual cache when a new query enters the system. This will allow us to know if any of the alternative plans is now more beneficial. Finally, we need to examine the algorithm for an arriving query to exploit the virtual caches.

We address the first issue by looking at the possible alternative plans which are kept along with the original intended execution plan. Note that this affects only the plan generation phase when a query arrives. As mentioned above, we will only be keeping k number of alternative plans. In our plan generation, we have only considered segmented right-deep plans. We adopt a two phase approach:

- In the first phase, we adopt a greedy algorithm similar to that proposed in [85] to generate a segmented right-deep plans. Recall that building relations in each segment fit in the memory.

- Second, for each segment, we re-optimize it to keep the best k plans for the segment.

Figure 4.3 shows a plan with k possible alternatives where k is set to 3. T(i) is either a base relation or a resulting relation from a plan segment.

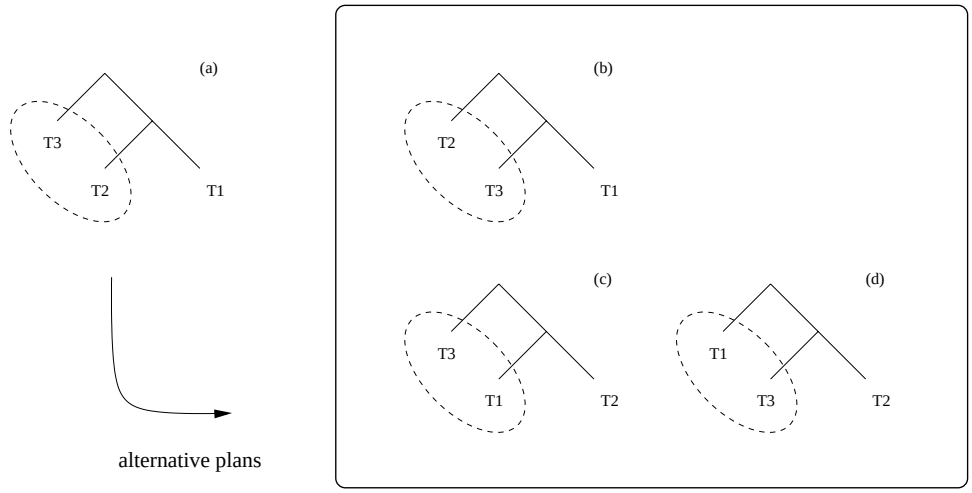
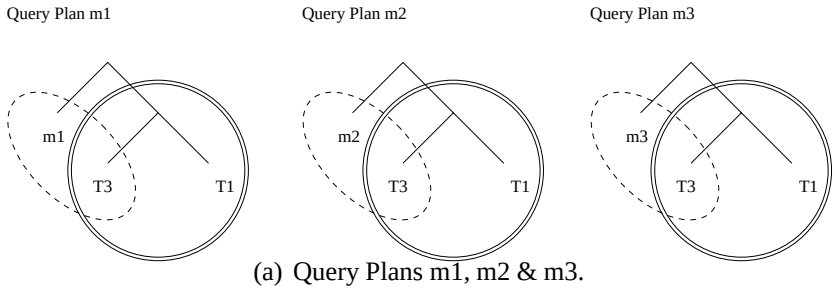


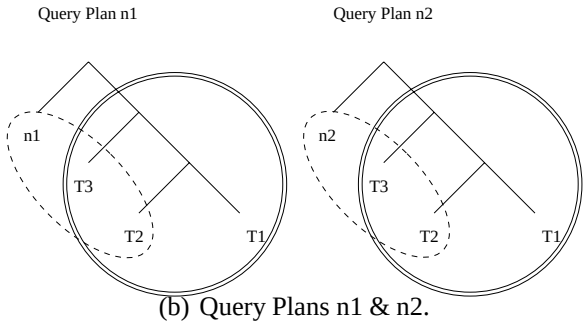
Figure 4.3: Query Plan T with Alternatives.

As an illustration, we observed that it will be beneficial to swap **query plan(a)** in Figure 4.3 to **query plan(b)** if the subsequent plans entering the system follow that as in Figure 4.4(a). The amount of benefit attained depends on the degree of salvaging possible. For this example, there will be a saving of 1 join operation and the corresponding I/Os needed for each of the queries in Figure 4.4(a) if a swap was made. However, it may also be possible that the original plan is more beneficial to the system if further query plans arrived following that as in Figure 4.4(b) where more savings is possible for salvaging larger sub-plans. Hence, it may happen that much work is done trying to improve the efficiency but work done may not necessarily leads to an improved performance. However, the chance of this happening is small if we consider that plan's occurrence appear with equal likelihood. So, the search space for finding common sub-plan increases

with more alternative plans kept. In this way, more salvages is possible with the search space opening up.



(a) Query Plans m1, m2 & m3.



(b) Query Plans n1 & n2.

Figure 4.4: Sequence of Query Plans

Next, we address the second issue by looking at the processing involved. k should be kept small for the processing to be efficient, but also not too small for this method to be effective. In our study, we have set k = 3. Each running plan or its alternatives has a benefit variable attached. This benefit variable is the total number of I/Os saved by other plans when 1 or more of its VCs (virtual caches) are used by these plans. When a new query plan arrives, all existing plans and their alternatives are searched. And for all plans with a matching VC, their benefit components are re-computed and their benefit variables updated. These computations consume cpu cycles but their benefits are realized

when huge amount of file I/Os are avoided when numerous plans reuse the sub-plans of these sub-optimal alternative plan instead of the optimal plan. The plan is swapped to its alternative whenever the benefit component of the alternative plan becomes more attractive or has a higher benefit value.

Finally, to generate a optimal plan for an incoming query, we adopted the two-phase optimization strategy. In the first phase, an optimized plan is generated. For simplicity, we also adopted the optimization algorithm in [85] to generate segmented right-deep plan in our work. We shall present only the post-optimization algorithm here. Figure 4.5 shows the post-optimization algorithm that is employed to exploit the virtual caches when multiple alternative plans are maintained. Essentially, the algorithm takes an optimal segmented plan as input, and returns the revised plan that reflects the reuse of relations. The algorithm is highly abstracted. It repeatedly examines a segment of the query plan QP. For each segment, it checks the subsets of relations that appear in the virtual caches (lines 6-17). For those that can be found in the virtual cache, it examines the alternative plans. At all times, it maintains the alternative plans that can provide the most benefit. Finally, QP is revised according to the alternative plans that can benefit most. At the same time, the plan QP is also updated to reflect any changes as a result of exploiting the virtual caches. The plan that is swapped is also revised accordingly. If there are more segments to be examined, the process continues; otherwise, the algorithm terminates and returns the revised QP.

Algorithm PostOpt

Input: Query plan of arriving query QP with segments $S_1, \ldots$
Output: Revised Query plan

```
1.    S ← nextSegment(QP)
2.    repeat
3.        maxBenefit ← 0
4.        R ← ∅
5.        A ← ∅
6.        foreach subset r of relations in S
7.            if there exists a virtual cache with relations of r
8.                for each alternative plan a
9.                    B ← determineBenefit(QP, r, a)
10.                   if B > maxBenefit
11.                       maxBenefit ← B
12.                       R ← r
13.                       A ← a
14.       RevisePlan(QP, R, A)
15.       UpdateProfile(QP)
16.       UpdateProfile(R)
17.       S ← nextSegment(QP)
18.   until S = ∅
19.   return QP
```

Figure 4.5: Post-processing optimization of plan for arriving query.

4.1.4 Reordering the Executing Segments with CoD

In our architecture, the **Scheduler** schedules a sub-plan to be executed each time the processor is free to accept a task. The **Scheduler** looks at the sub-plan at the head of the queue and gives it to the processor. In this section, we will study the effects on the overall system performance when these sub-plans in the queue are reordered. We have explored several heuristics to reorder these sub-plans. But in this work, we will be presenting only

2 ways in which these sub-plans can be reordered to benefit the overall system.

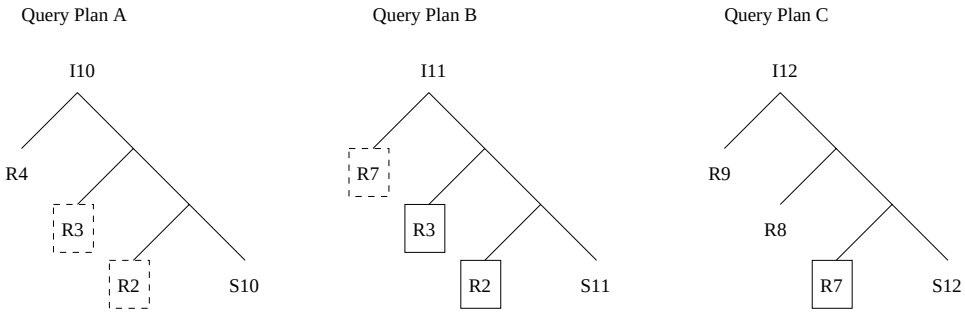


Figure 4.6: Reusing Base Relations.

The first method is based on the reuse of base relations to reduce the total number of system IOs. In this method, we try to place plans with common base relations adjacent to each other. In this way, these base relations could be reused by the succeeding plan and will not incur any additional IO for their use. As an illustration, Figure 4.6 shows a plan sequence where Plan B uses relations R2 & R3 of Plan A and Plan C uses relation R7 of Plan B. In this plan sequence, Plan B and Plan C will be able to save $|R2| + |R3|$ and $|R7|$ number of I/Os respectively. The reasoning is straight-forward, as we will retain all the base relations in memory which will be reused by the succeeding plan. Hence, these relations need not be brought in from the disk by the succeeding plan. For join involving these reused relations, that does not make use of the same join attribute as the previous plan, it will have to break the previous hash table and re-hash using the new attribute.

We have adopted a greedy approach in reordering these sub-plans. For every new sub-plan (denoted by Plan(new)) that arrives at the scheduling queue, we look in the

queue of plans and INSERT Plan(new) after Plan(i) when it gives the maximum benefit as follows:

max{{benefit of [Plan(i+1) - Plan(new)] + benefit of [Plan(new) - Plan(i)]} - {benefit of [Plan(i+1) - Plan(i)]}}

where

benefit of (Plan B - Plan A) is defined as the number of I/Os saved by Plan B which is the succeeding plan of Plan A, by reusing the base relations from Plan A. Since Plan(new) tries to look for matching base relations in all the plans in the queue, we have represented the presence of each base relation as a single bit in a bit string stored along with each subplan in the queue. In this way, the comparisons and the cost-benefit computations can be done quickly, allowing the reordering of plans to be carried our efficiently.

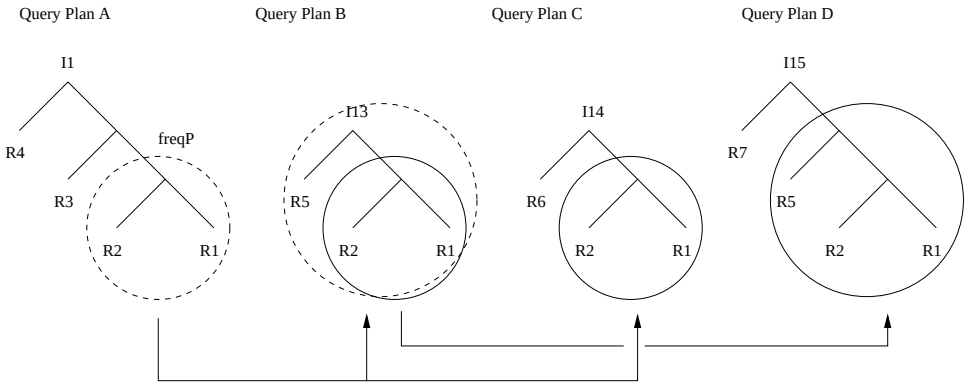


Figure 4.7: Priority Execution.

The second method pushes the execution of the most common sub-plan ahead. This method allows the execution of more plans to advance faster, so in turn their results can

be made available for other plans needing these results. Generally, more blocked plans will be freed earlier and can complete their execution faster. In Figure 4.7, we see that by pushing the execution of sub-plan freqP of Plan A, we will be able to advance Plan B and Plan C. Plan B and Plan C then complete their execution with their final join operation and release their results to Plan D which also completes its final join. As plans complete early, buffers are freed and made available for the next plan in the queue. This example reveals the advantage of propagating results as quickly as possible to increase the likelihood of having more plans completing earlier and freeing resources back to the system pool. The availability of resources in turn allows for better plan scheduling and execution in improving the overall system performance.

Periodically, for each sub-plan in the queue, we look at the number of other plans needing the intermediate results (or Virtual Caches) from this sub-plan and compute the total savings. From these computations, we push ahead the sub-plan with the greatest saving, to the head of the queue. We note the time of this computation and make a mark at the end of the queue, so that the next computation will continue from where it stopped. These periodic computations can be carried out while the executing plan is concentrating on I/O operations.

4.2 Experiments

To study the performance of the three extended mechanisms of CoD, we conducted a series of experiments. The following strategies were implemented;

- **Pipeline**. This strategy supports the first mechanism on pipelining. It essentially allows the CoD framework to salvage pipelined plans.
- **MRPipeline**. This strategy integrates the first two mechanisms on salvaging pipelined plans and the exploitation of multiple alternative subplans.
- **MRPipelineReorder**. This strategy incorporates all the three mechanisms.

Comparing **Pipeline** with **MRPipeline** allows us to see how significant is the benefit that can be derived from keeping multiple alternative plans. Comparing **MRPipeline** and **MRPipelineReorder** provides insights on the benefit of reordering. We also measure the benefits of the proposed strategies against two other strategies, namely Scramble-CoD [90] and Non-Pipeline. Scamble-CoD has been shown to perform well in most cases among all proposed CoD-based schemes [90]. Non-Pipeline is also a CoD-based scheme, except that its plan is the same as that of Pipeline, but pipelining is not supported. The system tested ran on an 296MHz Ultra SparcII at minimum load.

The default settings of the experiments follow those used in Table 3.2, and the results are scaled by the elapsed time for Non-Pipeline.

4.2.1 Experiment 1: Effect of MPL

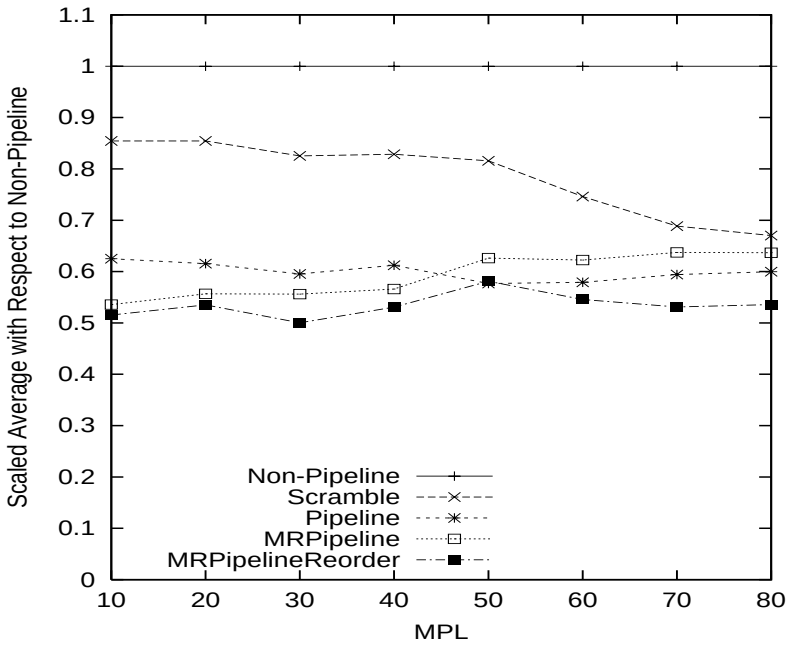
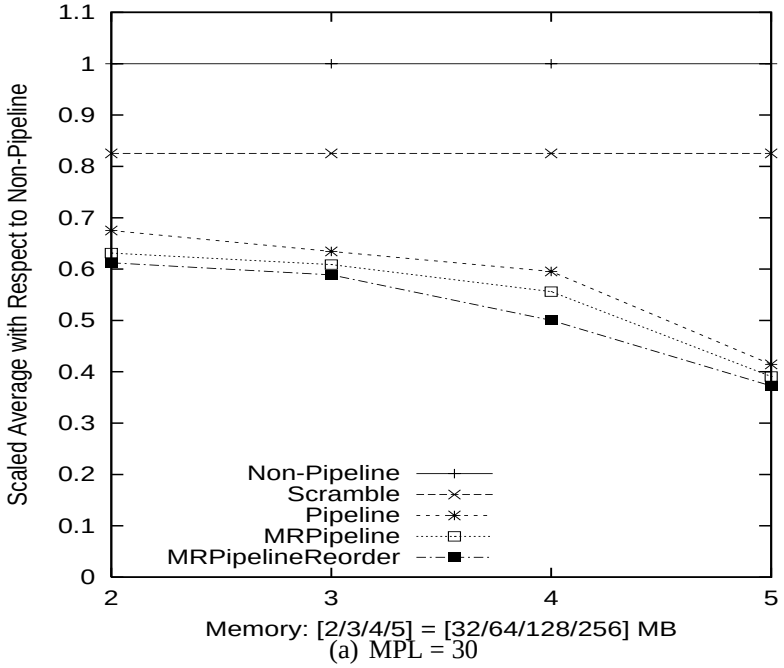


Figure 4.8: Varying MPL.

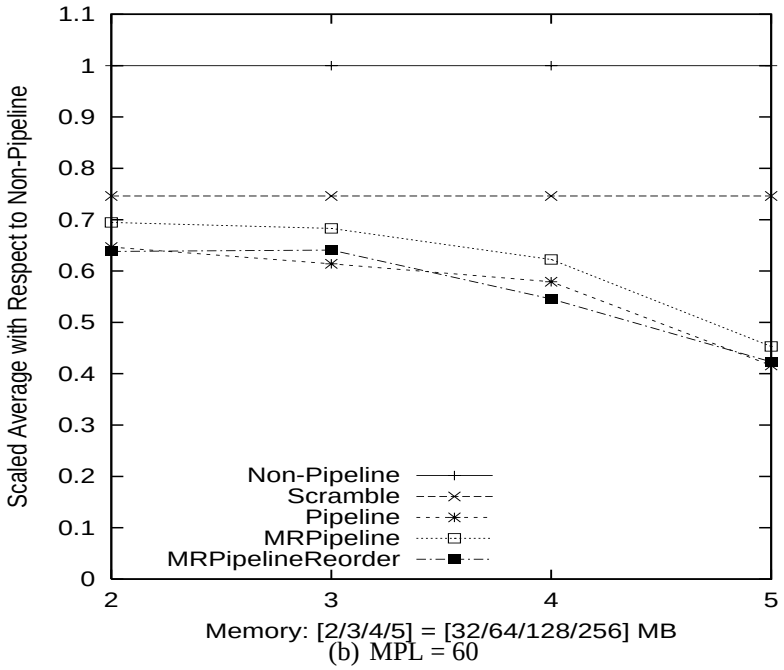
In the first experiment, we vary MPL from 10 to 80. The result is shown in Figure 4.8. First, we observe that the Scramble-CoD performed significantly better than Non-Pipeline. This is expected since plans generated by Scramble-CoD are typically bushy and are hence closer to the optimal plans than that generated by Non-Pipeline. On examining the three proposed algorithms, we notice that they performed better than both Scramble-CoD and Non-Pipeline. In fact, the gain over Scramble-CoD is as much as 30%. The reason for this better performance is attributed to the exploitation of pipelining, salvaging subplans for reuse and the better utilization of memory resources. For Pipeline, the result is as expected since the query plan operators are now executed on-

the-fly where intermediate results are not written out (unless they are cached for arriving queries). MRPipeline further exploits on salvaging common subexpressions from other plans by increasing the search space. However, it performs better than Pipeline when MPL is small and deteriorates when MPL is greater than 45. Upon investigation, we found that this is due to the greedy nature of our algorithm in exploiting alternative subplans. In our implementation, when we swap a plan, we remember only the benefit of the current subplan and not other subplans. For example, when a query Q_2 arrives, it may initiate a running query Q to swap its subplan S_1 to S_2; at a later time, another arriving query Q_2 may prefer Q's subplan S_1, but since the benefit of subplan S_2 in Q_1 is larger than the benefit of subplan S_1 in Q_2 (without considering the benefit of subplan S_1 in Q), subplan S_1 will not be picked.

Finally, MRPipelineReorder improves the utilization of available memory resources by retaining base relations for use by subsequent plans. Thus, we see that MRPipelineReorder has combined all these benefits, and performed best over the other two approaches. Comparing MRPipelineReorder and MRPipeline, we note that the gain by reordering subplans is only marginal (up to 7% at most). One possible reason is that we have adopted a greedy approach. We expect the gain to be more significant with a more effective scheduling algorithm.



(a) MPL = 30



(b) MPL = 60

Figure 4.9: Effect of Memory Size.

4.2.2 Experiment 2: Effect of Memory Size

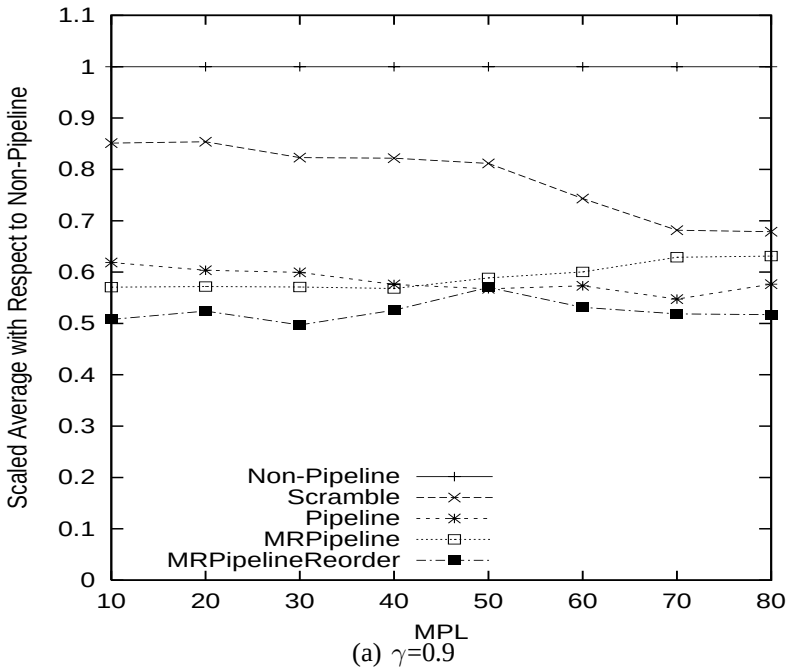
In the second experiment, we would like to study the effect of controlling the memory size by allowing pipeline plans with 2, 3, 4 and 5 joins. Figure 4.9(a)&(b)) shows the result of the experiment. We distinguish between two MPL values: 30 and 60.

First, we note that both Non-Pipeline and Scramble-CoD are independent of the memory availability since intermediate results are all materialized.[1] We note that as the memory size increases, performance of all three proposed strategies improve. This is as expected as more memory allows less intermediate results to be written out with a longer pipeline. The relative performance of the three proposed schemes remain the same as that reported in the previous experiment: for MPL of 30, MRPipelineReorder is the best scheme, followed by MRPipeline, followed by Pipeline; for MPL of 60, MRPipelineReorder is superior over Pipeline, which in turn outperforms MRPipeline.

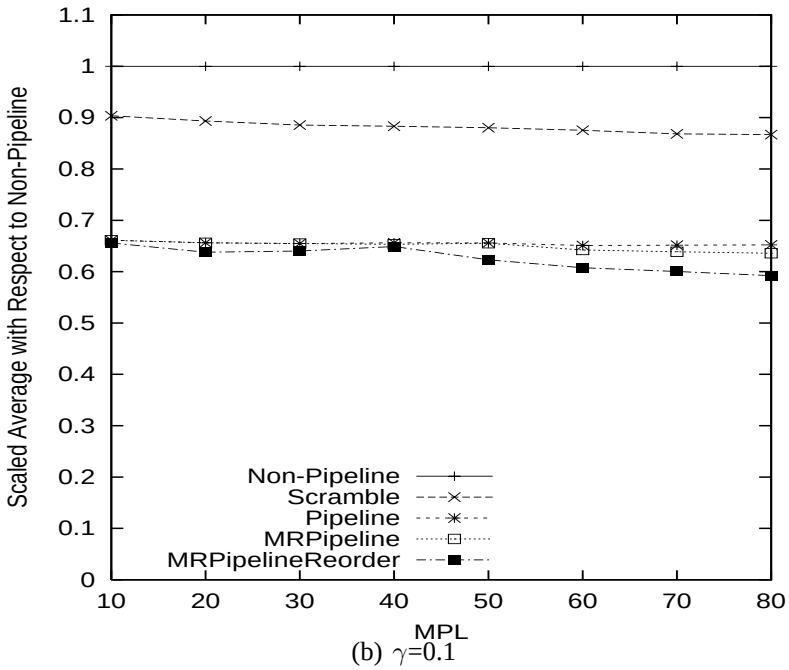
4.2.3 Experiment 3: Effect of γ, Q and N

In this experiment, we study the effect of the degree of overlap. This is essentially controlled by the parameter γ. We look at the two extreme ends of its values, 0.9 and 0.1 while varying MPL. The results for the γ values of 0.9 and 0.1 are shown in Figure 4.10(a)&(b) respectively.

[1] We note that Scamble-CoD may generate a different plan as memory changes, but in this experiment, we did not observe this effect.



(a) γ=0.9



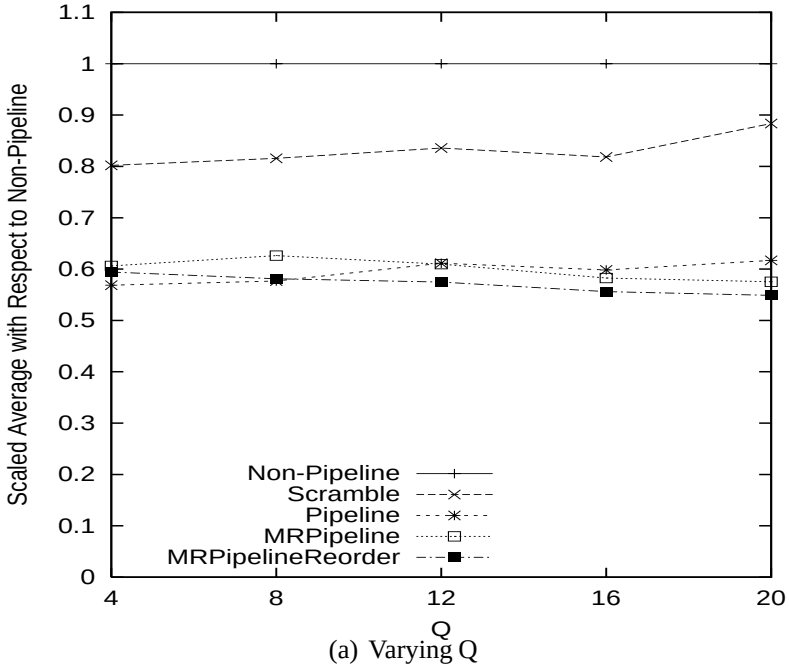
(b) γ=0.1

Figure 4.10: Effect of γ.

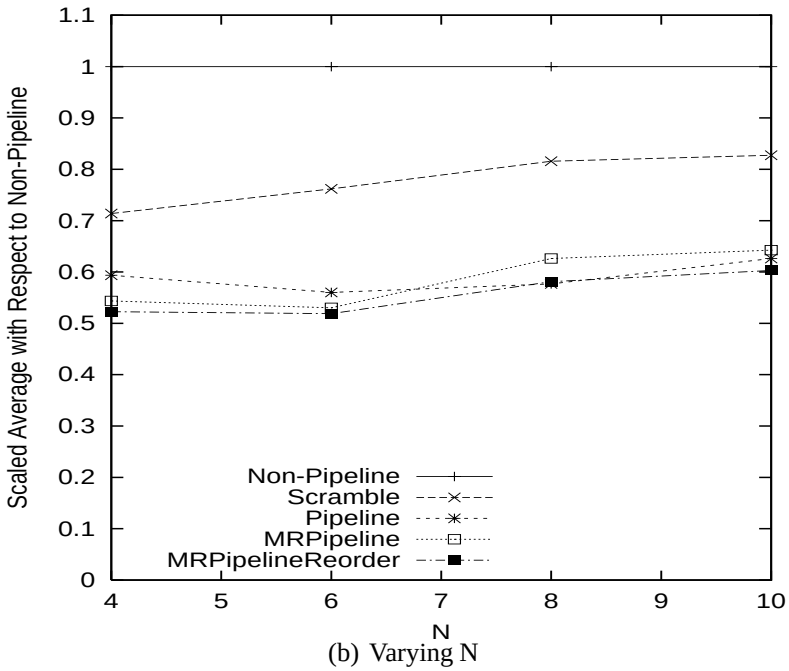
By having a larger γ, the probability of an incoming query belonging to the same basic queries increases. Hence, the amount of sharing of similar subplans between queries increases, leading to a significant gain in performance over Non-Pipeline (Figure 4.10(a)). However, as shown in Figure 4.10(b), for a small γ value, we observe that while the gain remains significant, it is not as much as that with a larger γ value. We also note that all the proposed schemes perform equally well for small MPL (< 50). This is expected since most of the queries are different, i.e., there is little opportunity for sharing, and thus the benefits of keeping alterative subplans and reordering diminish. However, as MPL increases, the opportunity for sharing increases. In particular, we note that MRPipeline is slightly better than Pipeline and reordering of segments can provide more gains.

Next, we study the effect of Q, by varying Q from 4 to 20. In Figure 4.11(a), we see that performance gain by the three proposed strategies is unaffected by the change in Q. This is so because the inprovement provided by these strategies does not rely on the probability of having chosen similar plans during plan generation, but on the exploitation of pipelining and better utilization of memory resources.

We also study the effect of N, the number of relations per query, by varying N from 4 to 10. We see that the performance dips for the Multi-Redundant strategies as N increases (See Figure 4.11(b)). The cause of such effect is due to the unpredictable amount of sharing of common subplans between queries as more combinations of generated plans are allowed with the relaxed number of relations in a query.



(a) Varying Q



(b) Varying N

Figure 4.11: Effect of Q and N.

To summarize, we see that the three proposed enhanced strategies remain superior over the Scramble-CoD and Non-Pipeline schemes, with the MRPipelineReorder-CoD approach performing best overall.

4.3 Summary

In this chapter, we have extended the CoD framework to further improve its performance. Three mechanisms were proposed: (1) to exploit pipeline plans, (2) to keep multiple plans, and (3) to reorder plans during execution. Our experimental studies showed that these mechanisms can lead to significant improvement in performance over CoD-based strategies. In particular, pipelining of query operators provide the most significant gain in performance.

Chapter 5

Cache-Wire

We have seen how the Cache-On-Demand caching techniques are able to improve query processing for multi-users in a centralized environment. These techniques are incorporated into the query system where we studied how query results can be selected for caching and be reused more efficiently. However, these techniques we have studied so far involve users that are not hindered by the data transfer latencies between them and hence communication cost was not included as a performance factor. Having studied how caching may benefit centralized query processing in the previous chapters, we turn now to study how it may be deployed for distributed context. In a distributed environment, the cost of communication contributes significantly to the performance of a querying system and generally requires considerable design attention. In what follows, we will look at how caching techniques are able to help in reducing the transfer overhead needed and the query response latency for querying in a distributed environment.

5.1 Introduction

Billions of queries are being circulated through the Internet each day, and reducing the query response latency has been the utmost priority in providing quality service in any business. This chapter discusses the opportunities and mechanisms to leverage distributed query processing performance using caches. In a multi-user environment, it is common for users to have similar and repeated queries. Consequently, these queries can be satisfied more efficiently by introducing caches for keeping copies of answers nearer to the users. The profusion in cheaper and greater storage spaces leads to more caches being made available at servers and clients, no matter what the quantity or speed used, and this has allowed the manifestation of algorithms to improve scalability, reliability and performance of the query engine. The use of web proxies to cache data for reuse has been a very effective solution to make the Internet more scalable for many years. Internet Caching Protocol (ICP) [42] was developed to allow web proxies to configure and access copies of the URL which are cached in other proxies. It was implemented in Squid [12], MS Proxy [69] and Cisco Cache Engine [29]. Queries are issued from one proxy to other preassigned proxies in order to determine the location of the Web-Objects. With more proxies assigned to each proxy server, more traffic will be generated. Similarly, for cooperative application-based caching at host site, flooding-based querying introduces a lot of traffic as more hosts are involved in the cache collaboration (Figure 5.1). Hyper Text Caching Protocol (HTCP) [41] is a later version of ICP and it incorporates the

capability of allowing hints to be provided from the proxy to the requester about Web Objects availability in other neighboring proxies.

Similarly, Content Distribution Network has been introduced to move business logic and data from servers closer to users focusing on minimizing the transfer delays. However, there are a huge amount of valuable data available for sharing and reuse on the Internet that remained largely unstructured. In our work, we have designed a framework to efficiently search these unstructured distributed data for reuse through selective routing with cache collaboration focusing on reducing the amount of message flooding. Collaboration is especially important for web searching in a P2P environment, as data resources and management in particular are distributed. We based our work on two observations on human behavior.

First, when we need information, we usually ask our friends around us. Interestingly, our friends usually remember what we asked, and would probably come back to us if they need the same information later (knowing that we may have had the information since we have previously asked for them). This leads us to the idea of caching (Requester Host-IP, [URL-Request|Files|Web-Objects]) pairs along the entire path (Hosts/Proxy-Servers) in which the request is forwarded. We assume that these hosts are making use of proxies that are chained or the host themselves caches the URL objects (in applications/web-browsers). In this way, when a request is received, a host can route the request more intelligently by directing it to other hosts that have made similar requests in the past.

Second, whenever we want to discard an item that is still usable (perhaps because we

are clearing our office, moving, or have no need of the item anymore), we usually would pass it to a friend (or even charity organization) who have need of it. This prompted us to introduce the idea of a host checking with other hosts before it trashed out the URL objects. This is particularly beneficial if the objects are computationally expensive to produce or the communication overhead to retrieve it from the distant source is high.

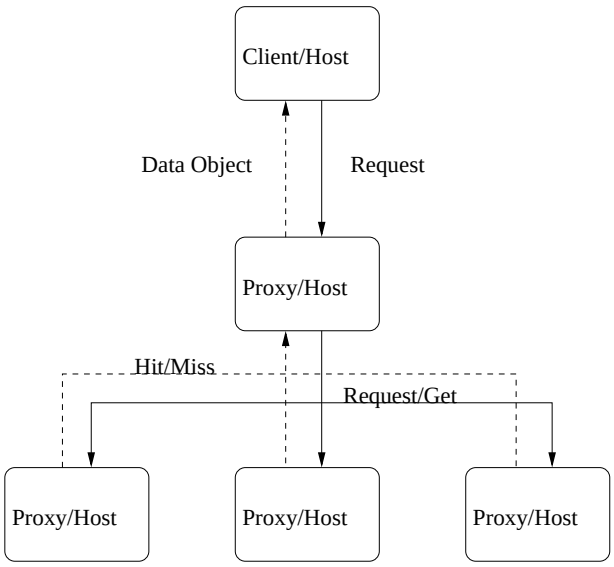


Figure 5.1: Request Forwarding.

We propose a framework called CacheWire that incorporates the two features to benefit the entire host network rather than the host itself. Moreover, we also design a graph-based scheme to facilitate host to predict if they need certain data. Our extensive performance study shows that CacheWire can lead to more successful retrievals in a much lower communication overhead and response time.

The rest of the chapter is organized as follows. Section 5.2 presents the architecture

of CacheWire. Section 5.3 reports the results obtained from our performance study. In Section 5.4, we examine the applicability of CacheWire using OLAP queries. Finally, in Section 5.5, we summarize the 2 mechanisms behind CacheWire, their performances and benefits.

5.2 CacheWire

In this section, we shall present the CacheWire framework which aims to improve the use of web caches in proxy servers or host caches. These caches store web documents/objects that could be reused by the neighboring hosts which could avoid, for example, going through the bottleneck of the network to other continents or going to a congested server. CacheWire distinguishes itself from existing routing and caching strategies. It applies our observations on human behaviors to support collaborative caching and routing to benefit the system as a whole rather than self. More importantly, it requires minimum overhead. In particular, it is designed to support the following:

- *Request Favored-Routing.* Request forwarding to all other hosts generates a lot of traffic in the network. Hence, it would be good if there is a way to direct the requests to specific host rather than blindly flooding the network. Requests can be cached as they passed through the hosts. If the request matches with one of the recently cached requests that passed through, it will be likely that the corresponding host already has the result. Hence, making use of this knowledge, routing this

request to those hosts which had recently made similar request seems promising. Even if these hosts did not retain the required web objects, they might have a good idea as to where these objects can be found nearby.

- *Cache salving.* In a host cluster, if the cached object is popular within a group of neighboring hosts, caching it at that point of high demand yield performance. However, the interest for this cache may shift from one group of hosts to another. In anticipation of such shifts, it will be beneficial if we manage the hosts' caches as inter-dependent mobile units within the region. We would like a strategy that enables us to move some of the valuable cache to a location that is near to the host that requires it instead of trashing it away.

5.2.1 The Architecture

Figure 5.2 shows the architecture of the *CacheWire* with the components to support the sharing and collaboration in the network. We assume a multi-session single user host where requests are issued sequentially (by the **Request Generator**) at regular interval in each session. The full arrows in the figure show the request path whereas the dashed-arrows show the cache-salvaging path. The network and user interfaces are omitted from the figure, but they work to connect the host to the network and for the user to issue requests respectively. We shall first describe the components before giving the overview on how the components interact.

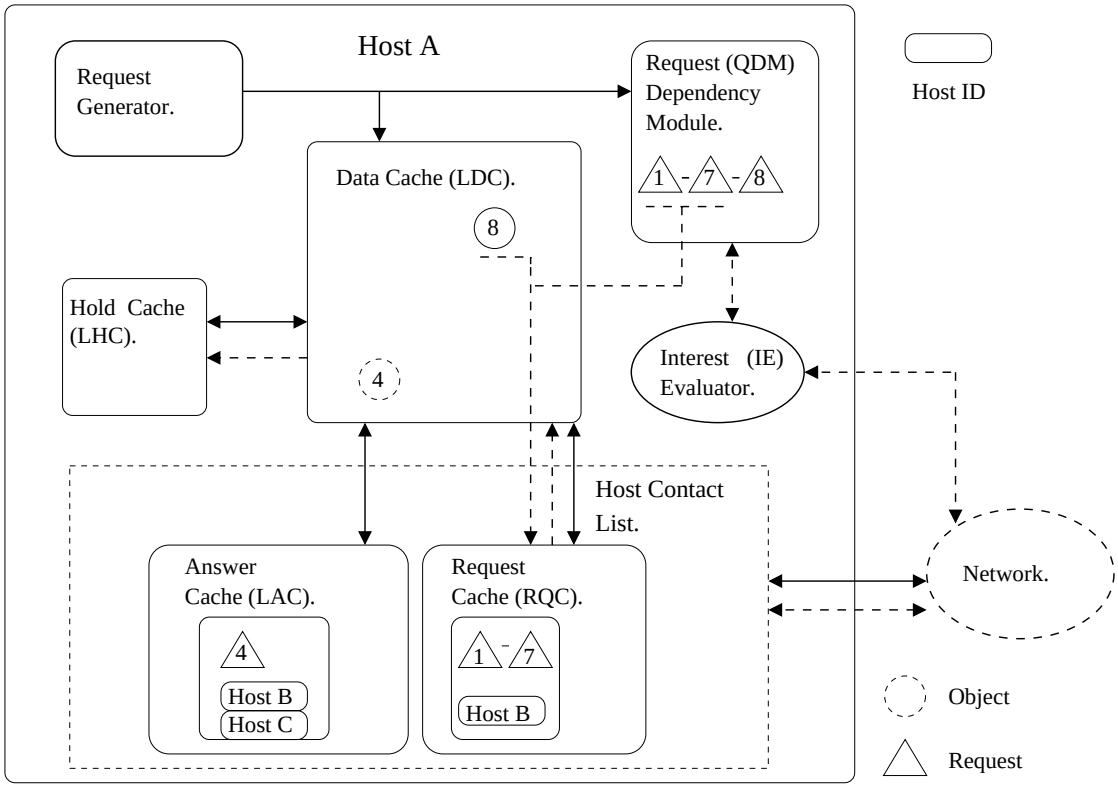


Figure 5.2: Architecture.

Data Source (DS)

The **Data Source** holds the persistent data object (URL source) belonging to the host that it is sharing with the other hosts. More generally, it also represents any data object that can be generated from this host (to be shared to the community).

Local Data Cache (LDC)

The **Local Data Cache** contains objects that are cached in the local host (from past requests). The cache values can be updatable or fixed and will be tagged correspondingly

depending on the value type. Fixed cache values will be like mp3 which are atomic and have content that are well described by their names. As for updatable values like database objects, their names might describe different values at different time. For such values, we have adopted the time expiration policy to determine the freshness during retrieval. There are many popular replacement policies like LRU-k (Least Recently Used), LFU (Least Frequently Used), LBF (Least Benefit First), and any one of them can be adopted by the cache manager. However, in our work, we make use of the VEGDSP algorithm [58] to implement the dynamic aging mechanism to avoid cache pollution by previously popular objects. The cache manager module is not shown in the Figure 5.2, but it is assumed to oversee the operations of all the caches.

Host Contact List

The **Host Contact List** keeps track of (Requester-Host-IP, URL-Request)-pairs of hosts who have issued request in the past. There are 2 structures currently built on top of this contact list. First, the **Local Answer Cache (LAC)** structure refers to the list obtained from requests issued by the host. As such, the Requester-Host-IP component actually points to hosts that have the result to the request. The other structure is the **Remote Request Address Cache (RQC)**. This list contains (Request-Host-IP, URL-Request) pairs where Request-Host-IP is the remote host whose request has been forwarded to the local host during the search process. As such, following our first observation (in section 1), it is likely that the remote host have some information about the request -

even if it does not cache the web object, it may know which hosts have the results. This cache is linked up virtually with other caches to facilitate the search for the requested objects. Periodically, when the host's load and the traffic is low, a message will be sent to the oldest cache location, to check the content's availability. Thus, the Host Contact List enables the system to make intelligent routing during requesting - as long as a forwarded request is similar to a past request, it can be directed to previous hosts that have issued that request.

Request Dependency Module (QDM)

The content of the cache comprises of objects that may have dependency (or clustering) relationship between them. For simplicity, we will only look at cache objects belonging to the same application when determining the dependency characteristic. We assume that each cache object has a corresponding request. Next, we define a request dependency list as a sorted list of requests in its arriving order. A user may have numerous request dependency lists, one belonging to each session, and these lists can be merged to form a directed graph. Hence, if we have one user per host and each host has a graph of this kind, we could have common sub-graphs between the hosts. This will allow us to track access patterns and look-ahead what are the possible next accesses in one graph given information derived from other graphs. In Figure 5.3, if we assume that most users behave generally in the same way most of the time, then graph1 will indicate that graph 2 is quite likely to ask for item 7. For instance, host that accesses a particular URL will

most likely click on one of the sub-links under it. If a sequence of sub-links starting from that URL is especially popular, then many hosts may have similar access trend. Following this argument, it is highly probable that many URLs have similar access trend at least for the initial few click sequence. In the literature, we observed other trend approximating techniques that are based on prediction, proxy-access affinity clustering or traffic pattern regularity [57, 97, 22].

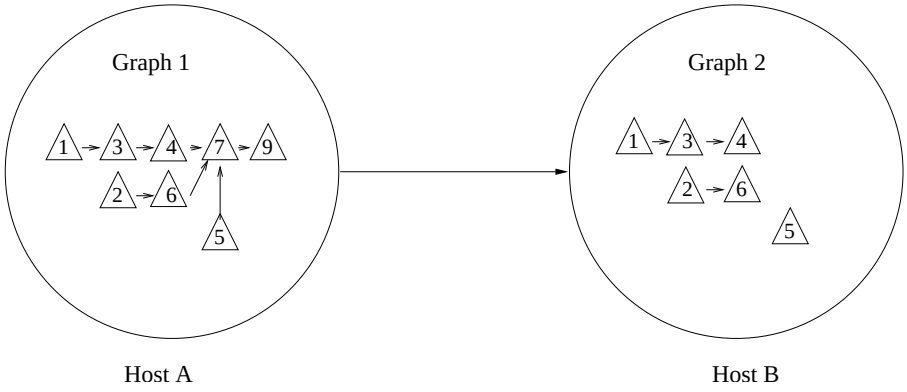


Figure 5.3: Probability of Occurrence.

Interest Evaluator (IE)

In our second observation (in the introduction), we have argued that it may be beneficial to the system if a host should pass its cached objects to other hosts that may need it (instead of simply discarding them). Upon receiving such a request (from the *offering* host that is trashing the object), the **Interest Evaluator** will determine if it needs the object in the near future and whether it is beneficial to itself. There are many possible

evaluation strategies that can be used. We adopt a simple heuristics that is based on the graph-based meta-data maintained in the QDM. We will discuss this in greater detail in Section 5.2.4. If the host finds the object beneficial, instead of requesting for the object to be transmitted to its local cache, it will request the offering host to retain the object for a short period of time (see Section 5.2.1).

Hold Cache (LDC)

In Section 5.2.1, if some host finds the object to be discarded useful, the offering host will retain it for a short period of time. This is an optimization that we have adopted to minimize unnecessary communication overhead - if the prediction is wrong and the object is transferred but not used, then we incur the extra communication overhead. On the other hand, when the object is requested during the period, then the offering host can simply discard the object after transmitting it. We introduce the **Hold Cache** for this purpose - objects in hold cache will be discarded whenever they are requested, or after the period of time to be held in the Hold Cache expires. The Hold cache shares the same set of memory slots from the main memory pool, hence, retaining more cache objects in the Hold Cache will mean less space available for the main Data Cache.

5.2.2 Request Favored-Routing

In CacheWire, whenever a request is issued, the host will first look into its Local Cache, and the Hold Cache (see full arrows). If the data is not available locally, it will send out

a request message to the neighboring hosts to assist in the search. A successful search in a neighboring host will return information with the source host address and download rate. Connection is then initiated and the download starts. At the same time, the Local Answer Cache is updated.

If the data is not available locally, then the request is searched against the Host Contact List and is forwarded to the list of hosts that have issued similar request in the past. If no similar requests have been found, then the request will be forwarded to other hosts. As an illustration, assuming we have 2 hosts as shown in Figure 5.4. When Host A issues a request q1, it first looks into the local data cache (LDC) to check if the result can be found. If not, it will then look at the remote request cache (RQC) to see if there is any remote request that has recently requested for the same request result. Consequently, a matching request is found with an associating host, Host B. A request message is then forwarded from Host A to Host B requesting for q1. Host B upon receiving the request will look into its LDC and response with an acknowledgment message. Finally, a successful handshake will allow Host A to start downloading from Host B the required request objects.

In the RQC, each request will be associated with a list of hosts and additional information regarding the request timestamp, host bandwidth size and host connection availability. When there are more than 1 host that have recently requested for the same request, this information can be used to assist in the selection (favoring a lower cost of successful retrieval) of the most favorable host to forward the request raised later by

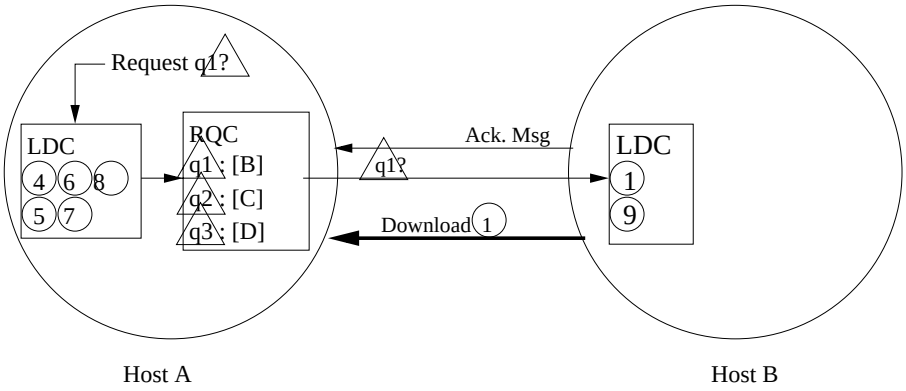


Figure 5.4: Favored-Routing.

the local host. This cost is calculated using a weighted sum of the factors taken from the information available at the local host. Given a request *q1*, a timestamp value *tq1B* (larger timestamp for more recent request) for this request associating with remote host B, bandwidth *wB* (larger value for higher bandwidth) available from host B and host B's connection availability at *cB* (ranges from 0 to 1 where 1 indicating always available), we can derive the cost of successful retrieval as $k1 \times (1\text{-}tq1B/t0) + k2 \times (1\text{-}wB) + k3 \times (1\text{-}cB)$ where where all factors are normalized to values within the range of [0,1], t0 is the current timestamp and k1, k2 & k3 are non-negative weights.

Sometimes, a request cannot be furnished by the cache of the remote host. This could be due to a few reasons. First, the cache size may be small and the remote host is very active. That is, the cache objects tend to be replaced much faster when generated objects are more beneficial than the cached objects. Secondly, the remote host may not have cached the object at all. This happens when the benefit of caching the object is low.

Lastly, it may also be possible that a remote host has yet to receive the object and hence unable to cache it.

So, what if the remote host cannot furnish the request, is there any other thing that the remote host can do? In fact, the answer is yes. It is the same as with humans, when we can't help someone with a request, and we are compassionate enough, we will try to reply with information on where would be the next better place to find the answer. For instance, a search through the remote node's LAC, may return a list of the remaining undeleted remote hosts that store the last downloaded object. Similarly, a search through the RQC, will return a list of remote hosts that might already have cached the objects. All these information are useful for the requesting host.

5.2.3 Piercing the Search Sphere

After a host issues a request to other hosts, it waits for a certain amount of time ($timeout$) for a reply before timing out. On the other hand, when a host receives a request and has no answer, it forwards it out to the other hosts. The timeout period limits the extent to which the request will be forwarded to. With all hosts functioning this way, a considerable amount of traffic will be generated. This slows down the effective propagation time of the requests and hence limits the effective reach of the request. Moreover, the neighbors of each host may not be close to one another physically. Hence, routing the request from neighbor to neighbor can be less effective than to route it straight to the

targeted host.

For instance, if the sum of the total propagation time between these neighbors is greater than the time to connect to the targeted host plus the propagation time, then the latter becomes more efficient. However, there will be a need to maintain information regarding these propagation time collected through pings (to neighbors) and information exchanges between neighbors. This method improves the elapsed time and increases the reach for each query within a given amount of time, but incurs more messages in the network. Thus, it is used only to track a good set of targeted nodes with beneficial cache objects.

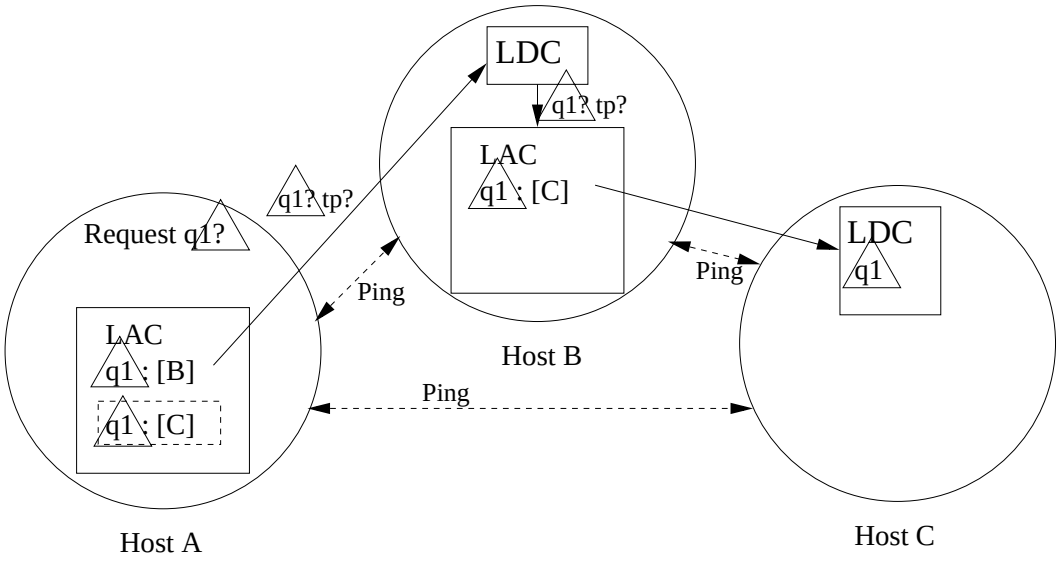


Figure 5.5: Updating LAC.

As an illustration, we look at a technique to do incremental improvement for the LAC. It is common for a host to request for a URL-page but never cache it for long, as it

may not be cost-effective. However, the host may retain some information regarding the source of download (from other caches) in the LAC. First, we look at how hosts collect the propagation time for a cache object. In Figure 5.5, at low load time, host A requests host B to check for the presence of q1 in his LDC. When host B detects an absence of this object, it checks it's LAC and returns the propagation time (host B to host C) which holds q1 (assuming host B had ping host C and stored the propagation time to host C. Host B returns a NAck(negative acknowledge) if it no longer has information on q1, and host A will remove the (host B, q1) entry in its LAC. Hence, with a positive reply from host B, host A now has the sum of the propagation time through neighbors to reach q1. Next, host A pings host C direct and compares $(t_p(AC)) + t_c(AC)$ with the sum computed previously. The connection time ($t_c(AC)$) is estimated to be $f + 2t_p(AC)$ where f is a positive constant and $t_p(AC)$ is the propagation time from host A to host C. If it shows that it is more beneficial to connect directly to host C than to pass through the neighbors, then (host B, q1) will be updated to (host C, q1) in LAC.

As the communication between the hosts increases, the hosts can be promoted from a benefit-collaborative level to a friend collaborative level for instance. This level of collaboration can be improved by increasing the resources allocated for sharing and lowering the benefit threshold for caching. The recommendation for promotion varies from host to host and this can be configured during the installation phase of the application software on each host.

5.2.4 Cache Salvaging

Generally, users access web pages through known URL or by simply following one of the links under it, with the latter being more convenient when one is exploring for details (aka web surfing). There can be no, few or numerous links under a particular page and it is not uncommon that only a few links are more popular than others within a page. For instance, the forum or the buy/sell pages belonging to a photography web page usually receive more hits than the member registration or the FAQ pages. CacheWire retains these lists of popular linked objects which are most likely required by others. As an overview, each starting URL-page leads to a few other popular pages and these pages will have a few other popular pages etc. It forms a tree (actually it is a graph but with cycles removed) for each starting URL-page. Hence, if popular pages are always just a few level deep along the access path, CacheWire will cache only the top few levels of each tree. In addition, we will also be considering popular pages that are with no sub-links. The following paragraphs explains the details.

Ideally, we would like to retain values that are heavy weight or expensive. We define a cache data as *expensive* when this data is downloaded from distant / low-bandwidth / transient host or this data is computationally heavy. So when a host decides to remove a data that is expensive (but not sufficient to remain in the cache), it should not be thrown away immediately, as it could benefit others at the cost of retaining it temporary. Nonetheless, this *expensive* cache can be totally useless if it is not interesting to anyone.

Also, how is it possible to justify that this cache is interesting to the other nodes? We define a cache data to be interesting if there is information showing that this data is in high demand from the neighboring hosts or if there is information indicating the likelihood of reuse by the neighboring hosts.

Assuming that a cache data is no longer needed or that it can always get it at a much cheaper cost then having to sacrifice another piece of data in the fully used cache, the data may be trashed in the next instance. Moreover, most hosts are transient in nature and caches are loss when hosts leave. Hence, we propose that the host that is about to trash it cache, should put in a certain amount of effort to prevent such loss. One way is to inform a group of neighboring hosts before trashing the set of cached data.

When we have a host that decides to trash its cache (see dashed-arrows), it will first route a message to a group of neighboring hosts asking if anyone is interested in its set of cache data. The message includes all the requests that generated the cache objects and several of the correlated requests leading to those requests. The message transmission can be optimized using similar technique as proposed in Cache Digest [76] which was introduced to allow proxies to make available cache content in a compact form. With this information, the receiving host predicts how interesting this set of cache data is and decides if he wants the initiating host to retain those values. The initiating host waits for a reply for a tolerable duration. If there are favorably many requests, the host will retain the values for a second time duration. Within this second duration, a neighboring host may indeed require the cache objects and request for a transmission. In the event when

no reply was received either in the first or second time duration, the cache objects will be trashed. In addition, the host does not guarantee that it will fulfill any wait duration if it needs to cache new objects and the cache is already full.

As an illustration, we assume that our cache contains cache objects for requests q4, q5, q6, q7 & q8. For each cache object, we increment the frequency count freqQi for each remote request made. If the cache object is initiated by the local host, we maintain a list of previous queries leading to each cached object submitted from the same application of the local host. These lists can be stored separately if we allow redundancy of replicated sub-list or assume that each item in the lists is independent of one another. But in practice, multiple dependencies between the items across the lists do exist. In this case, a dependency graph is used instead (see Figure 5.6).

For simplicity, the dependency lists (d-listQi) or graph (d-graph) leading to the cached object comprises of requests taken from the local hosts only. In the message format, graphs can be represented using a dictionary vector object (the mapping of the request strings to internal numbers) and edgeList object (the edges of the graph in number pairs).

As an illustration to show the collaboration between two hosts as shown in Figure 5.7, Host A has decided to trash its cache object for q7. It sends a message comprising of q7, freqQ7 and d-listQ7 to Host B to ask if it is interested in q7's cache. Host B upon receiving this message begins computing q7's benefit value for its use. Host B keeps a list of *n* previously requests for each application in the QDM. The benefit value for q7 is equal to $w1 \times Similarity(local{:}appQList, d\text{-}listQ7) + w2 \times Affinity(local{:}freqQi,$

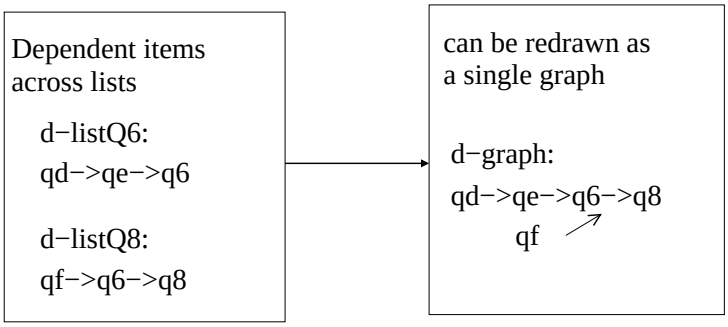


Figure 5.6: Dependency Lists and Graph.

freqQ7) where w1 & w2 are non-negative weights.

Similarity() returns a larger value when d-listQ7 matches a longer subsequence (or subgraph for d-graph) and when the subsequence is used by more applications. The value returned by Similarity() shows the extend of access pattern similarity between Host A and Host B. The subgraph isomorphism problem (finding common subgraphs in two given graphs) is proved to be NP-Complete. However, our problem is much reduced as we are given the starting vertices (the list of n requests leading to the request $q7$ whose cache object is to be trashed) for both graphs and the matching subgraphs must have the same vertices. For each request q_i leading to $q7$, we look for the longest matched path of predecessors and compute the number of vertices in the path ($longPathq_i$). The Similarity() measure of the graphs is then computed as the weighted sum of all the matched paths: $\Sigma_{i \leq n}(\Sigma_{j \leq longPathq_i}(\Sigma_{k \leq j} d))$ for the list of n requests, where d is a non-negative weight.

The Affinity() returns a percentile position as compared to the rest of the cached ob-

ject. This value shows the popularity level of the cache object within the neighborhood. It is computed based on the number of hits on the cached object in Host A. When Host A sends Host B an Affinity() value of a cached object, Host B compares this value with the rest of it's cached object and determines an estimated ranked position in the cache. Host B uses this position to decide if the cache object is interesting to him. Since Host A and B are neighbors, it is likely that they belong to the same neighborhood. Hence, if this cached object in Host A is well liked by other hosts, then it is quite likely that it will be well liked by others if placed in Host B.

The benefit value shows how interesting this cache object is to Host B and if it is sufficiently high, Host B will return an Ack Msg to indicate its interest for q7's result. Host A upon receiving this Ack Msg, will wait for a certain prearranged time for a confirmation to transmit. Within this time, if Host B indeed requires q7's result, then a request will be sent to Host A to start the download. Otherwise, Host A clears the cache when the time expires.

5.3 Experiments

To study the benefits of the collaboration and sharing in the proposed architecture, we conducted a series of experiments. The simulation is based on a system of cooperative application caches or a group of web proxy servers.

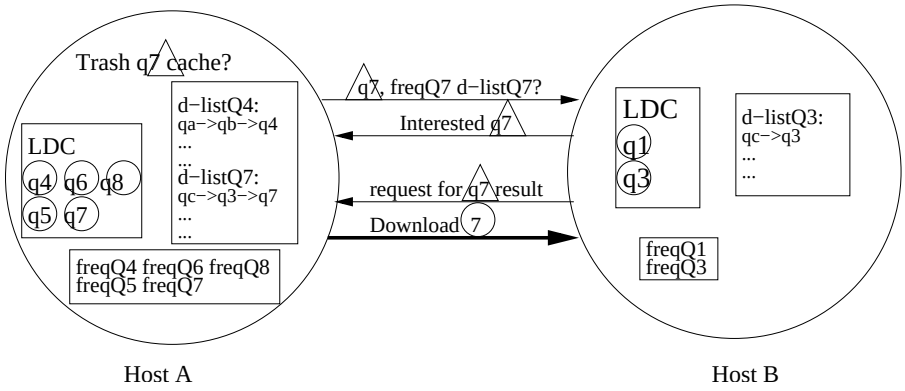


Figure 5.7: Salvaging Cache Data.

5.3.1 Experimental Setup

Table 5.1 shows the main parameters used. In this experimental study, each component of the proposed CacheWire is assessed. We have modeled the network topology based on the power-laws [31]. In simple words, a few hosts have a high-degree (connected to many others) and many hosts have small-degree. Each request initiated by the host will have a TTL (time-to-live) value that specifies how many hops it can traverse. For request generation, each host triggers off a lookup request based on its allocated query inter-arrival time $QUERYARR$. The $TRASHARR$ parameter sets a few random hosts to flush their data caches for handling a different application following a different requests pattern. These random hosts will then rejoin their initial clusters after some $REJOIN$ duration. Each host is created with a fix amount of cache memory, limited by M and is shared by the different caches each using a certain percentage of it. In the experiment, the total number of requests generated depends on the total number of hosts as well as

Notation	Meaning	Defaults
Network Parameters		
$HOST$	number of hosts	50
$NEIGHBOR$	number of neighbors	5
$ADDRSIZE$	host address	4 bytes
TTL	time-to-live value	3
$LINKDELAY$	average link delay	13 ms
Host Parameters		
$QUERYARR$	query inter-arrival time	10 ms
$TRASHARR$	cache trashing inter-arrival time	20 ms
$REJOIN$	cluster rejoin delay	300 ms
Data Parameters		
$OBJID$	object ID size	50 bytes
$OBJSIZE$	max object size	50 MB
Cache Parameters		
M	memory size	500 MB
Other Parameters		
$SIMTIME$	total simulation time	100000 ms

Table 5.1: CacheWire's Experimental Parameters.

the number of simulated time $SIMTIME$. To compare the amount of cost saved for each caching policy, we make use of a measure similar to that presented in [47].

$$DetailedCostSavingRatio_{time}(DCSR)$$

$$1 - \frac{\Sigma_i(E[q_i] + T_{miss}[q_i])}{\Sigma_i T_{total}[q_i]}$$

where

$E[q_i]$: $elapsed\ time\ to\ find\ Web\ Objects[q_i]$

$T_{miss}[q_i]$: $time\ to\ retrieve\ missing\ Web\ Objects[q_i]$ $using\ the\ URL$

$T_{total}[q_i]$: $time\ to\ retrieve\ ALL\ Web\ Objects[q_i]$ $using\ the\ URLs$

q_i : $request\ i$

5.3.2 Evaluation of CacheWire's Components

In the following series of experiments, we will present the results collected from six different CacheWire options as seen in Table 5.3.2.

Option	Components
NoCache	none added
LDC	LDC
LDC_RQC	LDC, RQC
LDC_LHC	LDC, LHC, QDM, IE
RQC	RQC
CacheWire	LDC, RQC, LHC, QDM, IE

Table 5.2: CacheWire Options.

The plotted values for the options are normalized against the values from the No-Cache module. For all the options, we have included the LAC module for routing queries.

5.3.3 Experiment A series: Effect of Varying Different Cache Size

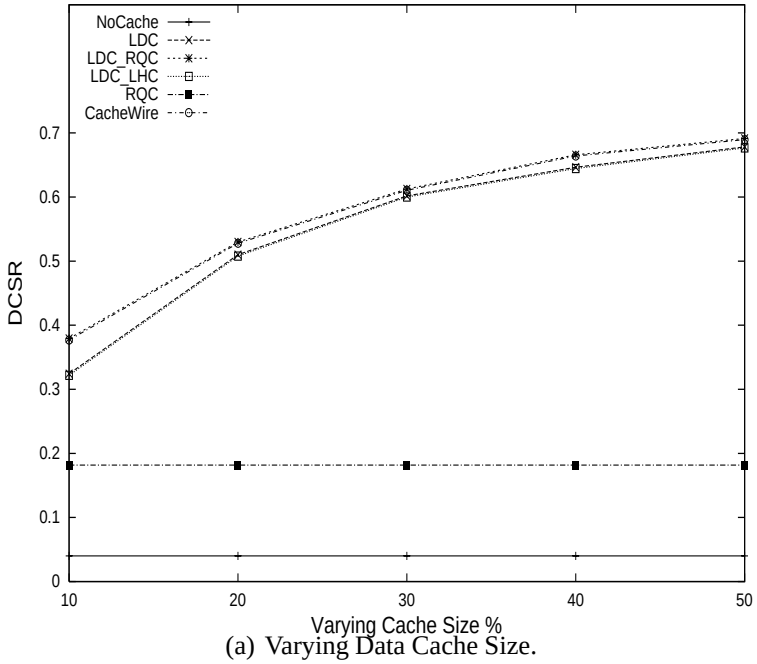
In the the first experiment, we vary the data cache (LDC) size from 10% to 50% of the available space. The result is shown in Figure 5.8(a). The result is as expected for increasing cache size since having more data in cache generally leads to an increase amount of savings. The figure also shows that when all engine modules are incorporated, it only performs marginally better than the rest, as having a very large cache (relative to the number of shared objects) have superseded the need for other modules. We also observe that the CacheWire module and LDC_RQC perform best among all the options

with the former being marginally better than the latter. The RQC performs the worst compared to the four options (other than the NoCache module) as expected since it contains no cache at all. The LDC_LHC performs only slightly better than LDC as performance of the former depends greatly on the benefit-threshold set, query arrival distribution, query pattern similarity matching algorithms etc.

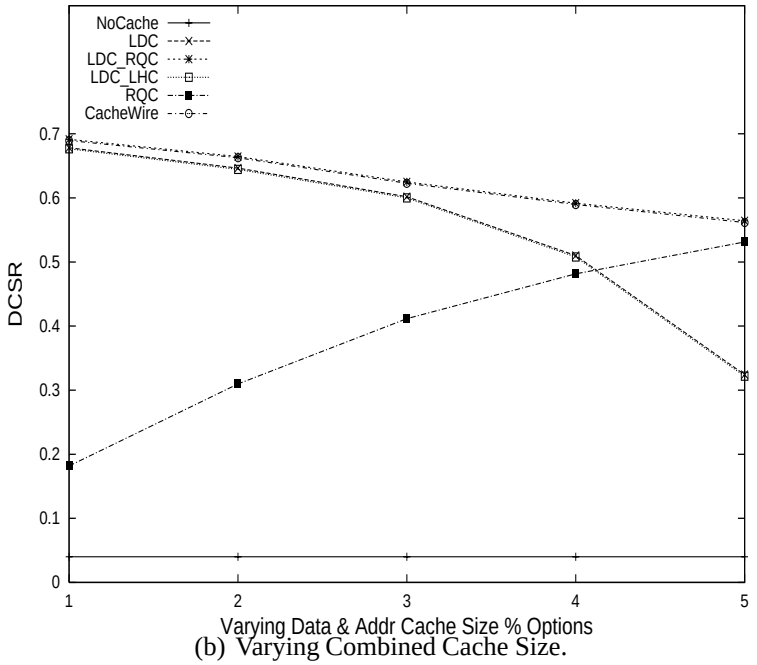
Next, we vary the combination(data, addr) cache (LDC & LAC) size from (50%,10%) to (10%,50%). In Figure 5.8(b), we observe that all options with the LDC module have a drop in their performance (i.e. the number of successful retrievals decreases and the elapsed time increase). Whereas, all options with the RQC module have either an increase in performance or a slower decrease in performance. Hence, for situation when large size memory is not available and data transfer is fast across network, we would recommend the use of these memories for caching the answer instead of using it for the data.

5.3.4 Experiment B series: Effect of Number of Peers and their Neighbors

In this experiment, we vary the number of peers from 20 to 100. The number of queries is correspondingly increased as more peers means that more queries are generated per peer. The number of web objects is increased as well since each peer provides some amount of shared information. The result is shown in Figure 5.9. It shows that performance



(a) Varying Data Cache Size.



(b) Varying Combined Cache Size.

Figure 5.8: Varying Data & Query Cache Size.

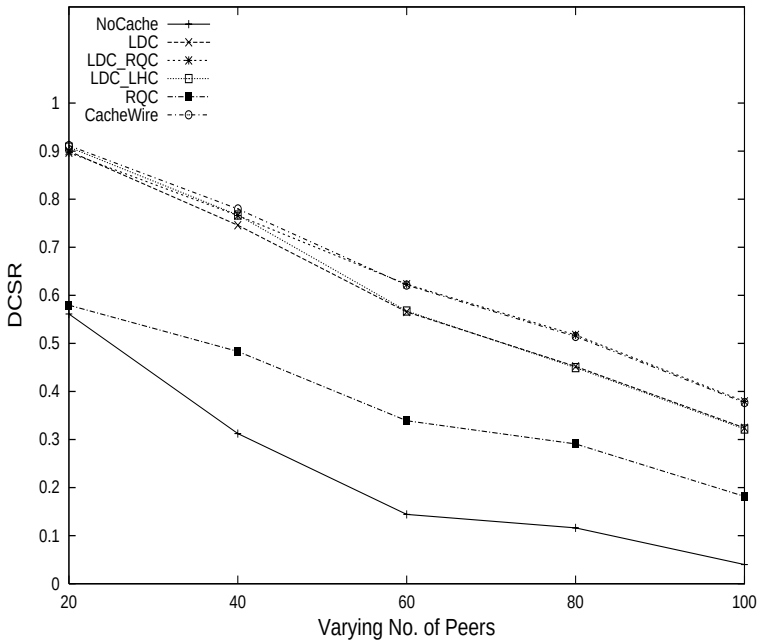


Figure 5.9: Varying Number of Peers.

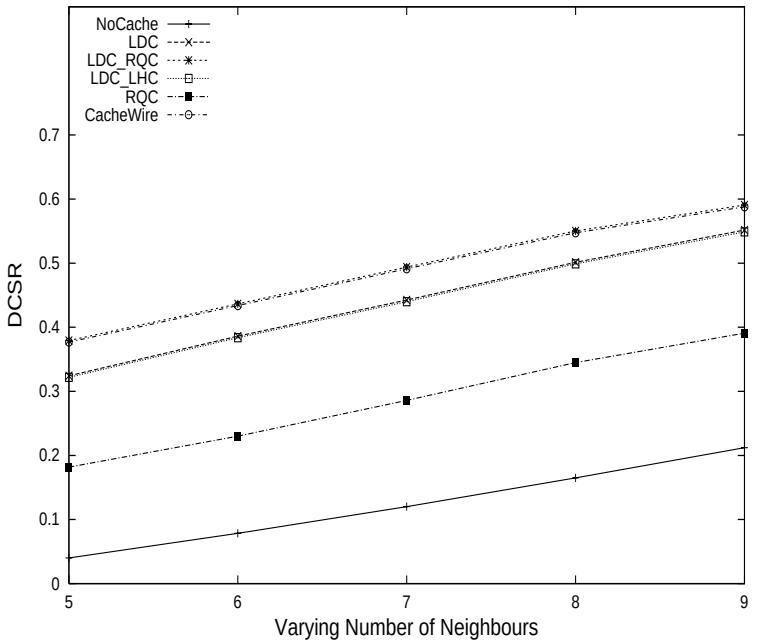


Figure 5.10: Varying Number of Neighbors.

decrease with increasing number of peers. This is because, the TTL set could only reach so far, hence, no matter how many caches you have in the network, it becomes less useful. Moreover, with more peers, more irrelevant data will be inserted.

In the second experiment, we vary the number of neighbors per peer from 5 to 9. The result is shown in Figure 5.10. Generally all options have increasing number of successful retrievals for increasing number of neighbors per peer since larger number of neighbors would mean a larger search space.

5.4 Applicability with OLAP Queries

In this section, we conduct performance study using OLAP (On-Line Analytical Processing) queries and make use of caches holding OLAP queries and the results. OLAP database system has been used as a tool for business data analysis and their queries demand quick response time in spite of being complex. Moreover, data updates are infrequent or could be scheduled in these systems which made it ideal for caching its results.

Caching has been proposed and implemented by OLAP systems in order to reduce response times. In [81], a data warehouse cache manager was presented which caches the query results together with the query string. The cache replacement and admission algorithms make use of a profit metric, which considers the average rate of reference, its size, and the execution cost of the associated query. In [28], a chunk-based caching ap-

proach was proposed. It allows queries to partially reuse the results of previous queries which they overlap and chunk miss is handled by a new physical organization for relational tables known as chunked files. Another cache manager was proposed in [52]. It uses multidimensional range fragments as the basic logical unit which provides a finer granularity for materialization. Similar to these caching strategies, CacheWire is used to deliver successful result retrievals, reducing overhead and improving response time with the use of query and fine granularity data caching.

In [47], an OLAP query caching approach in a P2P network was proposed. It constructs a large virtual cache by sharing the content of individual caches and works toward benefiting all peers. A voluntary caching policy was proposed. The caching policy attempts to exploit under-utilized resources that may exist in some peers and at the same time avoid wasting any result that has been obtained from the warehouse. When a peer with a full cache is unable to insert another entry with benefit lower than any of the entries in the cache, it will ask whether any neighbor wants to cache it. In CacheWire, the cache salvaging policy differs with this policy in many aspects. First, the item to be given away is from the cache and not a new entry. Secondly, each item to be given away has an accompanied trend information meant to assist neighbors to make item-acceptance decision. Thirdly, we do not just give the item to all neighbors but only to neighbors that potentially have use for them.

5.4.1 Dissecting OLAP Queries

Next, in this section, we present the adaptation of CacheWire for an OLAP system over a P2P network. Figure 5.11 shows a general layout of the OLAP system over a P2P network. The solid lines represent the peer connections whereas the dashed lines represent additional direct connections to the OLAP servers. P_2 and P_4 are the OLAP servers connected to the data warehouses (DW_1 and DW_2).

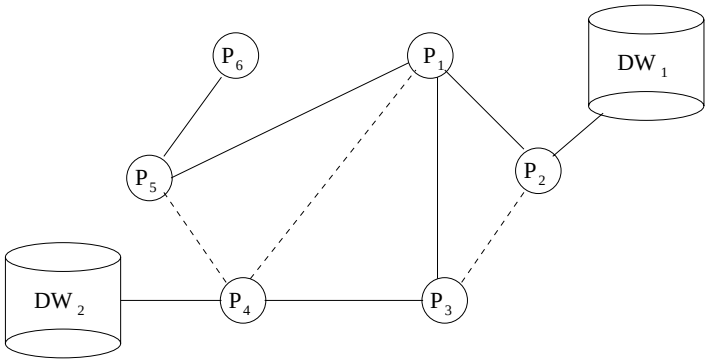


Figure 5.11: P2P OLAP System Network.

As discussed in [28], OLAP queries are typically repetitive and follow a predictable pattern. The same data might be accessed repeatedly by the same user. Also, as a consequence of the presence of hierarchies on the dimensions, data members which are related by the parent/child or sibling relationships will be accessed together. In OLAP applications, rolled-up operations and drill-down operations are used for generalization and specialization of fact tables. The sequence of these operations are often repetitive and predictable in OLAP applications.

Multidimensional OLAP queries involve group-by operation and aggregation across different dimensions and hierarchies. Systems that make use of these queries normally require interactive and quick responses. But aggregation operations required by such queries are normally expensive. A way to improve the evaluation of these queries is to store the aggregated results into caches and reuse them. On top of using these caches to benefit self, we have also proposed a cache salvaging strategy to instill collaboration between different nodes to improve the utilization of the cache before it is being flushed.

For our study, the LDC stores OLAP chunks which are less frequently invalidated since warehouses have infrequent updates. The Detailed Cost Saving Ratio is similar to what was presented in Section 5.3, however the WebObjects are replaced with OLAP chunks.

$$DetailedCostSavingRatio_{time}(DCSR)$$

$$1 - \frac{\Sigma_i E[q_i] + \Sigma_i T_{miss}[q_i]}{\Sigma_i T_{total}[q_i]}$$

where

$E[q_i]$	:	$elapsed\ time\ to\ find\ chunk[q_i]$
$T_{miss}[q_i]$	:	$time\ to\ retrieve\ missing\ chunk[q_i]$ $from\ data\ warehouse$
$T_{total}[q_i]$	:	$time\ to\ retrieve\ ALL\ chunk[q_i]$ $from\ data\ warehouse$
q_i	:	$query\ i$

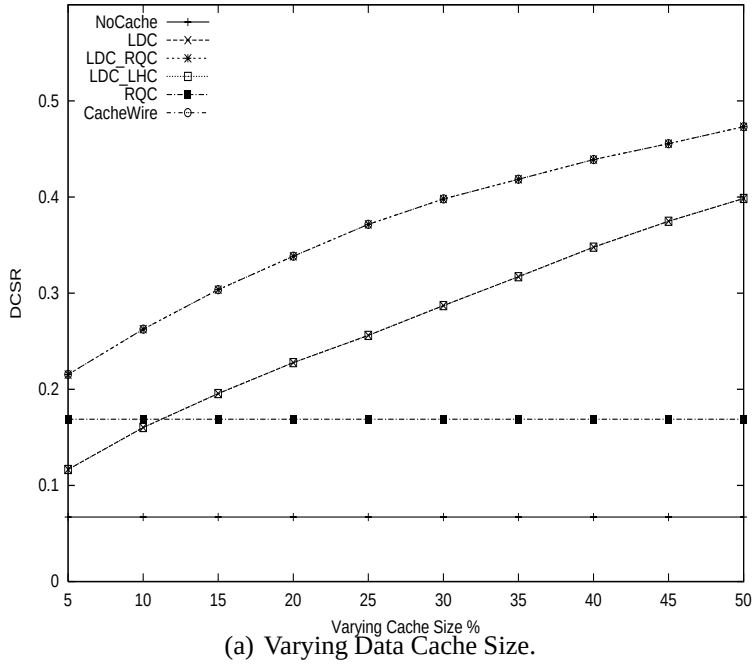
5.4.2 Experiments with OLAP Queries

In the experiments, the parameter settings follow that as in Table 5.1. The chunks were generated by uniformly chopping the range of each dimensions proportionally to the

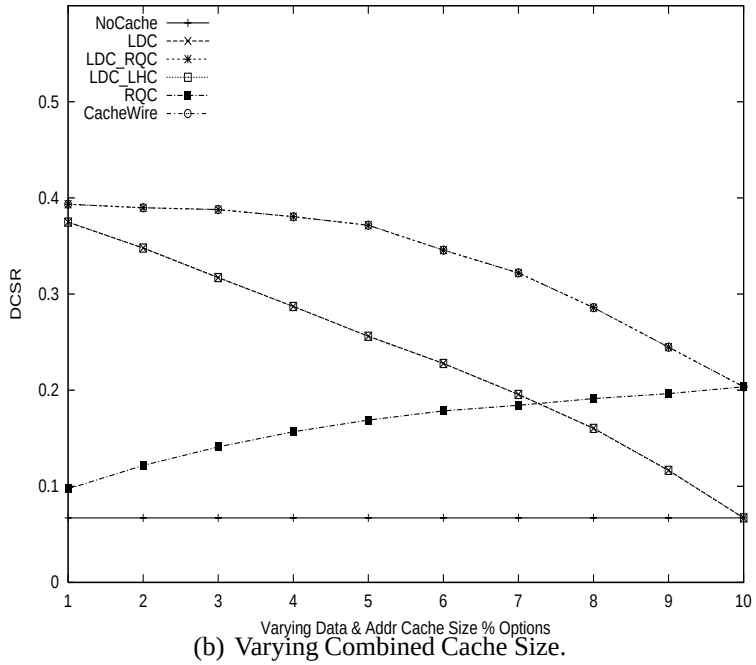
number of distinct values in the dimension. Each OLAP query is translated into a set of Chunk IDs and a search for these IDs is performed in the P2P network where missed chunks are retrieved from the warehouses. The results were similar to earlier experiments in Section 5.3. Figure 5.12(a) shows that increasing the cache size increases the performance of query retrievals. However, for small cache size, it will be beneficial to cache the queries of remote users. In Figure 5.12(b), we vary the cache and answer cache sizes. The result shows that performance drops as cache size decreases, but options with the RQC modules have better improvement with increasing answer cache. Next, in Figure 5.13, we see that all options decrease with increasing number of peers as more peers introduce more irrelevant data into the system. Lastly, in Figure 5.14, we see an improvement in performance with more neighbors introduced into the network. This is so because more neighbors contribute and share more cache spaces for matching queries. However, the performance stabilizes with more neighbors, as caching favors the popular chunks over the less popular, hence having more neighbors with each having cache of similar size, do not necessary mean that more chunks can be satisfied using the combined caches from the neighbors.

5.5 Summary

In this chapter, we have presented a framework call CacheWire to promote collaboration and sharing of data and information in the network. CacheWire allows information



(a) Varying Data Cache Size.



(b) Varying Combined Cache Size.

Figure 5.12: Varying Data & Query Cache Size.

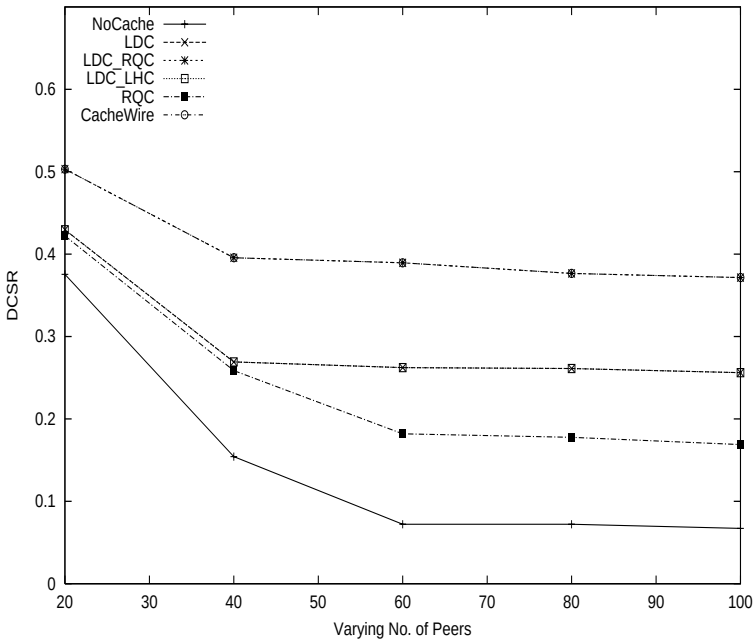


Figure 5.13: Varying Number of Peers.

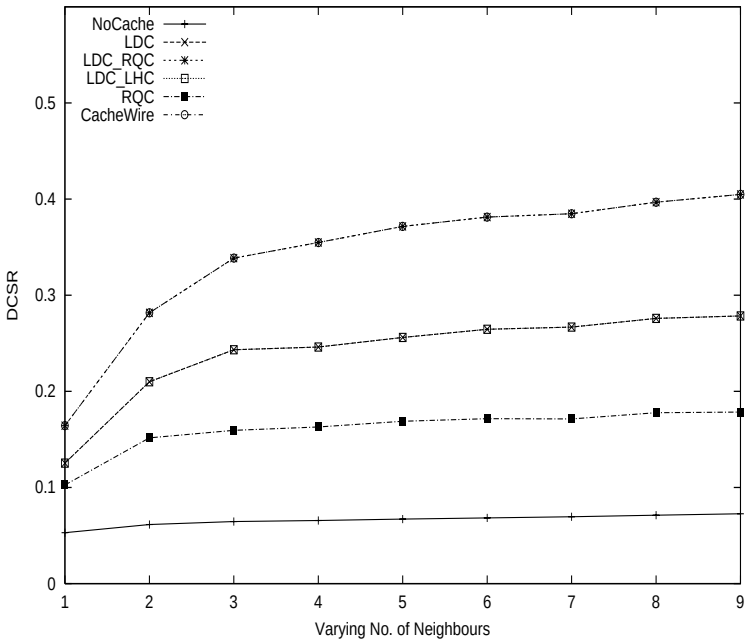


Figure 5.14: Varying Number of Neighbors.

to be gathered toward not only benefiting self but others as well. We have seen how this has allowed us to save cost during search and retrieval. We have also proposed a cache salvaging strategy to instill collaboration between different hosts to improve the utilization of the cache before it is being flushed.

In the experimental study, six different CacheWire's options have been evaluated for both WebObjects and Olap Queries. Our results showed that CacheWire's components generally contribute to a higher detailed cost saving ratio, and especially so for the RQC module when caching memory size is low. On top of striving for better distributed system performance, ideally, we aim to achieve a closed ecological system where all hosts/proxies are able to self-tuned and adapt to one another, yet not tightly coupled, in improving benefits for everyone.

Chapter 6

Cache-Coherence

So far we have looked at how caches can be used to benefit the processing of queries in different environments. The opportunities for the repeated use of the values in the caches have contributed in reducing the amount of processing needed otherwise. However, these values retained in the caches may vary from those at the source over time. Hence, these cache values must be updated when necessary in order to remain accurate and consistent with the source. In this chapter, we will look at a mechanism that efficiently maintains cache coherency between these caches and the source, illustrated using the server caches for edge computing.

6.1 Introduction

Many Web services are served from central locations, and could suffer from a number of bottlenecks ranging from Web and database server loads, to network delays. Server overloads can usually be alleviated through load balancing on a server farm. In contrast, network latency problems are usually out of the control of the Web service operators, as traffic to and from remote users have to pass through long-haul networks operated by multiple network providers that are often congested. Although aggressive build-up in recent years by telecommunication players has expanded the capacity of the long-haul networks, new technologies like Gigabit ethernet are making bandwidth much more affordable in the Metropolitan Area Networks (MAN). Given the relative price-performance of Wide Area Networks (WAN) versus MAN, the logical approach to reduce network latency is to push the Web services to the users, into the MANs.

Edge computing is being promoted as a strategy to achieve scalable and highly available Web services (e.g. [60]). It pushes business logic and data processing from corporate data centers, out to proxy servers at the "edge" of the network and within the MANs. There are several potential advantages: Running applications at the edge cuts down network latency and produces faster responses to end-users' applications and partners' Web services. Adding edge servers near user clusters is also likely to be a cheaper way to achieve scalability than fortifying the servers in the corporate data center and provisioning more network bandwidth for every user. Finally, by lowering the dependency on

the corporate data center, edge computing removes the single point of failure in the infrastructure, hence reducing its susceptibility to denial of service attacks and improving service availability.

In theory, edge computing is a natural extension of the Content Delivery Network (CDN) architecture. In practice, pushing application logic to edge servers introduces a number of technical challenges, one of which is data dissemination: For applications that run on a database, edge computing entails the distribution of (parts of) the database, to edge servers that perform query processing on behalf of the central DBMS. The accuracy of the query results produced by the edge servers thus hinges on how efficiently updated data can be disseminated to them. Proposed data dissemination techniques include eager versus lazy updates. Whichever techniques are employed, brute force data dissemination over WANs generates too much unnecessary network traffic and is not scalable, as we will demonstrate later.

The objective of our work is to improve the scalability of edge computing, through the introduction of an efficient data dissemination solution that propagates only updates to those portions of the database that are required by individual edge servers to satisfy their users and applications. Our proposed solution comprises two mechanisms – data scoping and delta profiling. *Data scoping* automatically monitors the active tuples and attributes that are targeted by the queries at each edge server, and uses this information to delineate the local data set that should be maintained at that edge server. Those data values within the data scope that are out-of-date and need to be refreshed are then iden-

tified by the *delta profiling* mechanism in logarithmic time (except when a substantial fraction of the key values in the table has been changed), all without requiring the master server to keep track of the data versions at any edge server.

The remainder of this Chapter is organized as follows. The edge computing framework, together with the proposed data scoping and delta profiling mechanisms, are introduced in Section 6.2. Following that, Section 6.3 analyzes the costs of the mechanisms, while Section 6.4 highlights some of the interesting experiment results. Finally, Section 6.5 summarizes the chapter.

6.2 Data Dissemination

The aim of this work is to improve the scalability of edge computing, by minimizing or even eliminating redundant updates to the edge servers. Our solution entails the construction of a customized index structure, called Verifiable B-tree (VB-tree), on each database table. We introduce a *data scoping* mechanism for tracking data values that are queried by user applications and need to be kept up-to-date at the individual edge servers, by having each server mark those nodes in its VB-trees that cover the result tuples for the processed queries. Furthermore, as each node in the VB-tree has an associated digest on all the data values in the subtree under that node, the digest of a marked node within an edge server's data scope can be compared to the node's latest digest to quickly profile the data updates that are relevant to the edge server – hence *delta profiling*.

6.2.1 Edge Computing Framework and Verifiable B-Tree

The Verifiable B-tree (VB-tree) was first proposed in [68] to generate verification objects for users to check the integrity of query results produced by edge servers. This section describes the edge computing platform, and also how the VB-tree is enhanced to support data scoping.

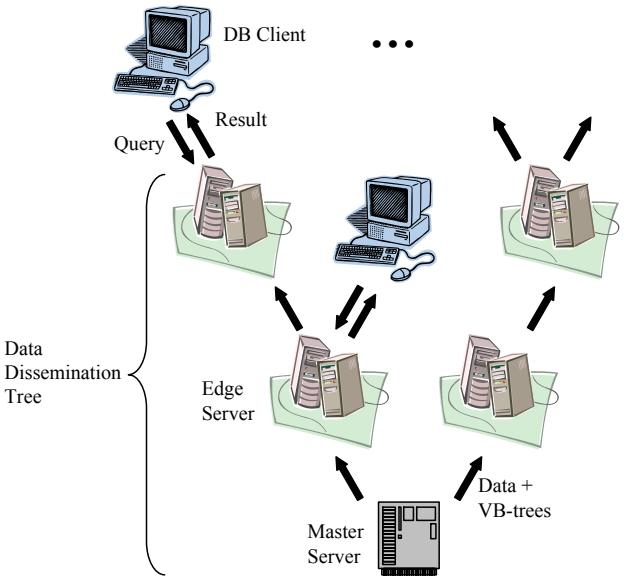


Figure 6.1: Edge Computing Set-Up

Figure 6.1 shows the set-up of an edge computing environment. In general, the servers in an edge computing platform can be organized in a dissemination tree with the master server as the root. The master server is responsible for creating and maintaining the VB-trees on the database tables, and for disseminating the data and VB-trees to the

edge servers to enable them to participate in query processing. This set-up enables the master server to delegate to the edge servers the authority to appoint additional edge servers, for example in response to high user traffic within individual MANs. In contrast to terminal servers that serve only their own queries, interior servers in the dissemination tree need to satisfy local requests as well as data demands from their child servers. Hence a flat dissemination tree limits the processing and storage overheads in data dissemination to a small number of interior servers, whereas a deeper dissemination tree spreads the overheads among a larger number of servers.

Figure 6.2 depicts the structure of the verifiable B-tree (VB-tree). It is constructed by adding digests to a dense B+-tree as follows:

- As shown in Figure 6.2(b), within each tuple a usage status U_{Ai} and a signed digest D_{Ai} are added for every attribute Ai, i.e., $U_{Ai} = 1$ if Ai has been included in a previous query result, and $U_{Ai} = 0$ otherwise. Furthermore,

$$D_{Ai} = k(DBname \mid tablename \mid tuplekey \mid Ai) \quad (6.1)$$

  where k is a one-way hash function on the concatenation of the database name, the table name, the key of the tuple, and the attribute value. A one-way hash function is one that is easy to compute but effectively impossible to invert [30]; i.e., if $b = k(a)$, then it is easy to compute b given a and k, but difficult to recover a from b and k. Popular one-way hash functions include MD5 [73] and SHA [1].

The attribute digests are used to compute the tuple digest D_T:

$$D_T \ = \ h(\sum D_{Ai}) \ \forall \text{ attributes } Ai \tag{6.2}$$

where h is another one-way hash function. D_T is then stored with the corresponding tuple pointer in the leaf node of the VB-tree. In addition, the tuple usage status is set to $U_T \ = \ U_{A1} \vee ... \vee U_{Am}$.

- For each leaf node N, a node digest D_N is derived from the tuple digests within it, i.e.,

$$D_N \ = \ h(\sum D_i) \quad \forall \text{ tuples } 1 \leq i \leq p \text{ in node } N \tag{6.3}$$

and the node usage status $U_N = U_1 \vee ... \vee U_p$. The node digest and usage status are stored with the corresponding child pointer in the parent node as shown in Figure 6.2(c). This process is performed recursively up the VB-tree.

- Finally, a digest and usage status are computed for the root node and stored as part of the metadata of the VB-tree.

6.2.2 Data Scoping

As mentioned above, the purpose of data scoping is to identify the local data set that is actively accessed by the user applications, and hence needs to be maintained at each

edge server. The data scope is the union of the scope of all the queries executed by that edge server, i.e., the tuples and attributes in the query results.

Definition: The **query scope** is the smallest subtree, within the VB-tree on the target database table, that envelops all the result tuples of a query (involving selection, projection or join operations).

To track the query scope, each node P that is found to cover at least one result tuple is marked, by setting $U_P = 1$, as the edge server traverses down the VB-tree during query execution. Figure 6.3 gives an example, with shaded pointers and circles indicating the marked nodes and result tuples, respectively. Within the query scope, some of the tuples or attributes may be filtered from the query result through selection or projection operations. This is why, in the subtree under the marked node N in the example, there is an unmarked node and some unmarked tuples. [1]

Definition: The **data scope** of an edge server consists of the smallest subtree, within the VB-tree on each database table, that envelops all the query scopes on that table.

Since the individual query scopes may contain unmarked nodes/tuples/attributes due to filtered tuples and attributes, the resulting data scope may also contain unmarked subtrees. When the edge server sends the data scope to its parent server to facilitate data

[1] In [68], the query scope is used to generate a verification object for the recipient to authenticate the query answer. The verification object contains the *signed* digest for the top node of the query scope, as well as signed digests for any unmarked nodes within the query scope. The recipient can then re-combine the data values in the query answer with the digests for the unmarked nodes, and check if the result tallies with the signed digest for the query scope.

dissemination, these unmarked subtrees can be left out to reduce transmission overhead. This is illustrated in Figure 6.4. Hence the information that is sent to the parent server includes:

- the portion of the VB-tree within the data scope, minus any unmarked subtrees;
- the path from the data scope up to the root of the VB-tree (for delta profiling which will be presented next); and
- the digest for the marked attributes within each tuple (*not* the actual data values of the marked tuples).

With data scoping, the edge servers no longer need to hold the entire database. Instead, each edge server can now store only portions of the database that fall within its data scope. Thus a data scope is essentially a materialized view of the database, and data scoping is a mechanism that automatically shapes the view based on data demand at individual edge servers. As for the VB-trees, the edge servers still have to cache the entire index structure to enable them to handle any queries that extend beyond the data scopes. However, only changes to the VB-tree nodes within the data scopes are disseminated. Depending on the selectivity of the data scope, these savings can potentially lead to a sizable reduction in storage and update traffic.

A second factor that affects the efficacy of data scoping is the frequency at which new queries are encountered. Specifically, the data scoping mechanism is likely to be effective when the target data set at an edge server is relatively stable, for example where

the queries are embedded within application programs. If the edge server also runs ad-hoc queries, it is likely to encounter queries that are beyond its data scope from time to time. When that happens, the edge server will merge the new query scope into the data scope, and submit it to the parent server immediately to initiate a data refresh. The parent server may in turn need to escalate the refresh request toward the master server.

Another consideration is whether data scoping as described above is really more efficient than a naive approach of sending triplets of ⟨tuple-id, attrib-id, digest⟩ to the parent server. While this will enable the parent server to detect existing tuples that have been modified, it will not catch *new* tuples that have been added to the database table. Consequently, each parent server will have to implement additional mechanisms to track the content or data version at the individual child servers. Finally, the edge server needs to obtain fresh signed digests for its VB-trees through the parent server anyway (because the edge server is not trusted to certify data), in order to use the VB-trees for query result authentication [68]. Data scoping also facilitates that by delimiting the parts of each VB-tree that needs updating.

6.2.3 Delta Profiling

Upon receiving the data scope of a child server, the parent server can simply send back all the data values within the scope. A more efficient alternative is to perform delta profiling to isolate those data values within the scope that have been modified: Starting with the

node B at the top of the data scope, the parent server compares node B's digest with the digest for the corresponding node in the local, up-to-date VB-tree. If the two digests match, there has been no update within the data scope. If the digests differ, the parent server proceeds to check the marked child nodes of node B, then to their child nodes in turn, until finally reaching the tuple attributes. Those marked attributes that have mismatched digests are the deltas within the child server's data scope. The complexity of this process is logarithmic to the size of the underlying table.

Figure 6.5 illustrates how delta profiling tracks down the tuples and attributes to refresh, using the data scope in Figure 6.4. The blackened pointers in the figure denote digests that are different between the data scope and the up-to-date VB-tree. The example shows that the parent server manages to narrow down to only some attributes within one tuple that need to be updated, which is a huge saving over refreshing the entire data scope. Moreover, the cost needed for delta profiling is only logarithmic in the number of tuples covered by the data scope.

Figure 6.5 also demonstrates that even though a node A within the data scope has a mismatched digest, all of the marked child nodes of node A can potentially have matching digests with their counterparts in the up-to-date VB-tree. In that case, the digest of node A was changed because of modifications to unmarked tuples/attributes. Those modifications can safely be ignored by the child server.

While update operations on non-key attributes of a database table lead only to changes in the node digests, insert, delete and update on key attributes could change the structure

of the VB-tree. As a result, some of the nodes within the child server's data scope may no longer have counterparts in the parent server's current VB-tree, as demonstrated in Figure 6.6(a). In fact, it is possible that there are so many changes to the underlying database table that even the counterpart for node B at the top of the data scope cannot be located within the current VB-tree. This is the scenario in Figure 6.6(b).

As the structure of the VB-tree could change over time, the range of search keys under each node has to be inspected as the delta profiling mechanism traverses from the root, down to the data scope, then on to the subtrees within the data scope. If, for any node, the mechanism fails to locate counterparts with matching key range in the parent server's copy and the child server's copy, a "subtree divergence" has occurred at that node. At that time, the underlying database table is divided into regions as shown in Figure 6.6, and refreshed as follows:

- Region A: There has been no changes in this region of the VB-tree structure, and the tuples here are excluded from the data scope. No action is necessary.

- Region B: This region contains tuples that are new to the child server, and possibly some tuples that already exist at the child server. Since the tuples are beyond from the data scope, however, there is no need to disseminate them to the child server. Instead, only the corresponding VB-tree nodes are sent. The digests within these nodes are set to an invalid value to trigger a data refresh if ever the child server executes a query that targets any tuples here.

- Region C: This region of the VB-tree is within the data scope, and has had structural changes. New tuples here should be disseminated to the child server, deleted tuples should be discarded from the child server, and updates to any existing tuples should be propagated.

- Region D: Tuples in this region have been deleted from the parent server. Instruct the child server to discard them.

- Region E: This region is part of the data scope, though there has not been any changes to the VB-tree structure here. Proceed as normal to match the node and tuple digests to identify any updated attributes that need to be refreshed.

The algorithm for delta profiling, including handling of changes to the VB-tree structure, is given in the next section.

6.2.4 Delta Profiling Algorithm

Global variabes:
path = [(root$_{min}$, root$_{max}$), ..., (DS$_{min}$, DS$_{max}$)];
// path from VB-tree root to data scope at
// the child server; each entry gives the
// range of search keys under the node

Ptree; // up-to-date VB-tree at parent server
Rtree; // copy of Ptree

node$_{diverge}$;
// last common node between parent server and
// child server before their subtrees diverge

```
1.  main() {
2.      copy Ptree to Rtree;
3.      node_diverge = NULL;
4.
5.      DeltaProfile();
6.
7.      if (node_diverge != NULL) {
8.          recompute digests in Rtree from leaf nodes up;
9.              // because delta profiling may have changed
10.             // some digests in Rtree
11.         send to child server the subtree under node_diverge
12.             in Rtree, and digests along the path from the
13.             root down to node_diverge;
14.     }
15.
16.     exit;
17. }
```

```
1.  DeltaProfile() {
2.      nodeP = root of Ptree;
3.      (P_min, P_max) = key range of Ptree;
4.      (C_min, C_max) = first key range in path;
5.
6.      if ((C_min, C_max) != (P_min, P_max)) {
7.          node_diverge = nodeP;
8.          Divergence(P_min, P_max);
9.          return;
10.     }
11.
12.     while ((C_min, C_max) != (DS_min, DS_max)) {
13.         (C_min, C_max) = next key range in path;
14.         if ((C_min, C_max) does not exist in nodeP) {
15.             node_diverge = nodeP;
16.             Divergence(P_min, P_max);
17.             return;
18.         }
19.         nodeP = child node from pointer between
20.                     C_min and C_max;
21.         (P_min, P_max) = (C_min, C_max);
22.     }
```

```
23.
24.     // nodeP at the parent server corresponds to the top
25.     // node of the data scope
26.     nodeC = top node of the data scope received from
27.                 the child server;
28.     CheckDataScope(nodeP, P_min, P_max, nodeC);
29.
30.     return;
31. }

1.  CheckDataScope(nodeP, P_min, P_max, nodeC) {
2.     // nodeP and nodeC are corresponding nodes within
3.     // the data scope, one from the parent server and
4.     // the other from the child server.
5.     // Both nodes have identical key range (P_min, P_max).
6.
7.     if (search keys in nodeP and nodeC are not identical) {
8.        // encounter structural divergence within data scope
9.        node_diverge = nodeP;
10.       MergeTuples();
11.       return;
12.    }
13.
14.    ptrP = first pointer in nodeP;
15.    ptrC = first pointer in nodeC;
16.    K_min = P_min;
17.    K_max = first search key in nodeP;
18.
19.    // region E in Figure 6.6
20.    while (ptrP != NULL) {
21.       if (ptrP.digest != ptrC.digest) {
22.          if (ptrP.child is a tuple) {
23.             // found an updated tuple within the data scope
24.             send to child server the attribute values of
25.                the tuple that have mismatched digests;
26.          } else {
27.             // the child node has mismatched digests
28.             CheckDataScope(ptrP.child, K_min, K_max,
29.                                ptrC.child);
30.          }
31.       }
```

```
32.         ptrP = next pointer in nodeP;
33.         ptrC = next pointer in nodeC;
34.         K_min = K_max;
35.         if (there is another search key in nodeP)
36.             K_max = next search key in nodeP;
37.         else
38.             K_max = P_max;
39.     }
40.
41.     return;
42. }

1.  Divergence(P_min, P_max) {
2.     // P_min and P_max denote the key range of the
3.     // parent server's subtree under node_diverge
4.
5.     if (P_min < DS_min) {
6.         // region B in Figure 6.6
7.         instruct child server to delete tuples with key
8.             in the range (P_min, DS_min);
9.         set all the digest of tuples in the range
10.            (P_min, DS_min) to "invalid" in Rtree;
11.    } else if (DS_min < P_min) {
12.        // region D in Figure 6.6
13.        instruct child server to delete tuples with key
14.            in the range (DS_min, P_min);
15.    }
16.
17.    if (DS_max < P_max) {
18.        // region B in Figure 6.6
19.        instruct child server to delete tuples with key
20.            in the range (DS_max, P_max);
21.        set all the digest of tuples in the range
22.            (DS_max, P_max) to "invalid" in Rtree;
23.    } else if (P_max < DS_max) {
24.        // region D in Figure 6.6
25.        instruct child server to delete tuples with key
26.            in the range (P_max, DS_max);
27.    }
28.
29.    MergeTuples();
```

```
30.     return;
31. }

1.  MergeTuples() {
2.      // region C in Figure 6.6
3.      merge the tuple entries in leaf nodes from parent
4.          server and child server with key in the range
5.          (DS_min, DS_max);
6.      while (merged list is not empty) {
7.          if (there are two entries with matching keys at
8.              the head of the list) {
9.              // one each from parent server and child server
10.             remove both entries from the list;
11.             if (digest of the 2 tuple versions do not match) {
12.                 send updated attribute values of the tuple
13.                     to the child server;
14.             }
15.         } else if (the first entry in the list is from
16.                     the parent server) {
17.             remove the first entry from the list;
18.             send the tuple value to child server for insertion;
19.         } else {
20.             // first entry in the list is from the child server
21.             remove the first entry from the list;
22.             instruct the child server to delete this tuple;
23.         }
24.     }
25.     return;
26. }
```

6.2.5 Eager versus Lazy Updates

The data scoping mechanism described above can be employed in conjunction with either eager or lazy update techniques. With eager update, a parent server notifies its child servers whenever there are data updates. (In practice, we expect the master server to

batch the updates instead of disseminating individual updates the they occur.) Once a child server sends back its data scope, the parent server identifies, from among the modified data, those updates that fall within the child server's data scope as presented in the previous section. The shortlisted updates are then returned to the child server together with the latest VB-tree. Since each batch of updates triggers an immediate propagation, there is no need for a separate delta profiling process to match the data scope with the VB-tree in order to discover which tuples have or have not been modified. The eager update protocol is summarized in Figure 6.7.

With lazy update, a parent server continues to notify its child servers whenever there are data updates. However, the child servers no longer respond with their data scopes immediately, but simply register the update notification. Only when a query arrives and there is an outstanding update notification, does a child server submit the current query scope as its data scope to the parent server for updates. The parent server then performs delta profiling on the data scope, and returns the new pertinent data values together with the latest VB-tree. Upon receiving the updates, the child server will then cancel the update notification. If the parent server itself has an outstanding update notification, it will in turn merge the child server's data scope into its own data scope, and send it to its parent server in order to receive updates. Therefore the query could experience a substantial delay whilst updated data flow down the dissemination tree. However, this sacrifice may be worthwhile for databases where updates are more frequent than queries, as only the latest data values are disseminated with lazy update. Figure 6.8 summarizes

the lazy update protocol.

Besides pure eager and lazy update models, it is possible to employ a combination of both techniques within the dissemination tree. The motivation is that, intuitively, eager update is more efficacious for edge servers that have high query frequencies, whereas lazy update may suffice for edge servers with infrequent activities. Consequently, upstream edge servers within the dissemination tree are likely to require eager updates because they are responsible for the data demands of their child servers plus local applications. In contrast, downstream edge servers that serve only occasional local users could benefit from lazy updates.

With such a hybrid model, a heuristic for choosing the propagation technique for a given edge server is to compare the relative frequencies of its queries versus data updates. If from the time of the last data update, the server has supplied the same data more than once (as evident from any overlap in the query scopes), either for local queries or to help its child servers with their queries, then the edge server is likely to warrant eager updates. Another heuristic is to limit the propagation delay, or the number of hops from any edge server to its nearest parent server in the dissemination chain that receives eager updates.

Parameter	Description
$\|DB\|$	Size of database (MBytes)
$\|DS\|$	Size of data scope (MBytes)
s_{data}	Selectivity of data scope = $\frac{\|DS\|}{\|DB\|}$
$\|US\|$	Size of update scope (MBytes)
s_{update}	Selectivity of update = $\frac{\|US\|}{\|DB\|}$
f_{update}	Frequency of updates
$\|QS\|$	Size of query scope (MBytes)
s_{query}	Selectivity of query = $\frac{\|QS\|}{\|DB\|}$
f_{query}	Frequency of queries
B	Block size (KB)
R	Record size (KB)
fan	Fan-out of VB-tree

Table 6.1: Cache-Coherence's Analysis Parameters

6.3 Analysis

Having introduced the data scoping and delta profiling mechanisms, we now evaluate their effectiveness in disseminating data from the master server to the edge servers. Specifically, using "standard" eager and lazy updates as baselines, we assess the relative performance of data scoping and delta profiling, deployed individually and in conjunction with each other.

The parameters for our evaluation, which will be explained as they are used, are summarized in Table 6.1. The key performance metric is the volume of network traffic generated in disseminating data updates. For lazy updates, we will additionally consider the propagation delay, defined as the average duration that queries are put on hold while their query scopes are refreshed.

For this analysis, we assess the impact of the various algorithms on a pair of parent-child servers for simplicity. We will measure the overall performance for the entire dissemination tree in the experiments later.

- Standard eager update

  With standard eager update, the parent server transmits the data updates and the modified VB-tree immediately. The size of the data update each time is $|DB| \times s_{update}$, whereas the size of the VB-tree is $\frac{fan^{h_{DB}}-1}{fan-1}B$ where $h_{DB} = \left\lceil log_{fan}\frac{1000\times|DB|}{R}\right\rceil$. Assuming that the initial update notification is small and can be ignored in this analysis, the network traffic is:

$$\begin{aligned} Tr &= (|DB| \times s_{update} + \\ &\quad \frac{fan^{h_{DB}}-1}{fan-1}B) \times f_{update} \end{aligned} \tag{6.4}$$

- Data scoping with eager update

  According to the protocol in Figure 6.7, each update cycle involves the exchange of the data scope, data updates, and updates to the VB-tree. The size of the data scope is $\frac{fan^{h_{data}}-1}{fan-1}B$ where $h_{data} = \left\lceil log_{fan}\frac{1000\times|DB|\times s_{data}}{R}\right\rceil$. For the VB-tree, since only updates that are within the data scope need to be sent to the child server, the overhead is roughly equal to the size of the data scope. The size of the data updates is $|DB| \times s_{data} \times s_{update}$. Hence the total network traffic is:

$$Tr = (|DB| \times s_{data} \times s_{update} +$$

$$2 \times \frac{fan^{h_{data}} - 1}{fan - 1} B) \times f_{update} \quad (6.5)$$

assuming that s_{data} and s_{update} are independent.

- Delta profiling with eager update

  The network traffic load is the same as for the baseline because, without an explicit data scope, all updates have to be included in the *delta*.

- Data scoping + delta profiling with eager update

  The network traffic load is the same as with data scoping alone, because all updates within the data scope are included in the *delta*.

- Standard lazy update

  With standard lazy update, the parent server accumulates the updates until the child server activates a refresh operation upon receiving a query. This requires the parent server to implement extra mechanisms to track the data version at each child server. Hence it cannot be compared directly with the other algorithms that are investigated in this chapter; we will therefore not consider standard lazy update any further.

- Data scoping with lazy update

  According to the protocol in Figure 6.8, the child server activates a refresh operation for each query. The data scope that is sent to the parent server at that time

is thus the query scope. As for the size of data update, without any version tracking mechanism the parent server will have to refresh all the data within the query scope. The network traffic is thus:

$$\begin{aligned} Tr &= (|DB| \times s_{query} + \\ &\quad 2 \times \frac{fan^{h_{query}} - 1}{fan - 1} B) \times f_{query} \end{aligned} \tag{6.6}$$

where $h_{query} = \left\lceil log_{fan} \frac{1000 \times |DB| \times s_{query}}{R} \right\rceil$.

- Delta profiling with lazy update

  Without an explicit data scope, the child server will have to send the entire VB-tree to the parent server. The network traffic here is:

$$\begin{aligned} Tr &= (|DB| \times s_{update} + \\ &\quad 2 \times \frac{fan^{h_{DB}} - 1}{fan - 1} B) \times f_{query} \end{aligned} \tag{6.7}$$

  where $h_{DB} = \left\lceil log_{fan} \frac{1000 \times |DB|}{R} \right\rceil$.

- Data scoping + delta profiling with lazy update

  With both data scoping and delta profiling, there will be a further reduction in update size over data scoping alone. The network traffic is:

$$\begin{aligned} Tr &= (|DB| \times s_{query} \times s_{update} + \\ &\quad 2 \times \frac{fan^{h_{query}} - 1}{fan - 1} B) \times f_{query} \end{aligned} \tag{6.8}$$

where $h_{query} = \left\lceil log_{fan} \frac{1000 \times |DB| \times s_{query}}{R} \right\rceil$, and assuming that s_{data} and s_{update} are independent.

The above analysis leads to the following observations:

1. For eager update, data scoping is beneficial if (4) $\geq$ (5), i.e.,

$$|DB| \times s_{update} \times (1 - s_{data}) +$$

$$\frac{fan^{h_{DB}} - 2 \times fan^{h_{data}} + 1}{fan - 1} B \geq 0$$

   Since $s_{data} \leq 1$ by definition and $fan \gg 2$ in practice, the inequality holds if $h_{DB} > h_{data}$. Hence data scoping is beneficial as long as the scope is not the entire VB-tree.

2. For lazy update, data scoping is more effective than delta profiling if (6) $\leq$ (7), i.e.,

$$|DB|(s_{update} - s_{query}) +$$

$$2\frac{fan^{h_{DB}} - fan^{h_{query}}}{fan - 1} B \geq 0$$

   Since lazy update makes sense only if $s_{query} \leq s_{update}$, and $h_{query} \leq h_{DB}$ by definition, data scoping is indeed the more effective of the two.

3. However, the combination of data scoping with delta profiling is still the best for lazy update, as (6) $\geq$ (8).

4. Comparing (5) and (8), since by definition $s_{query} \leq s_{data}$ and $h_{query} \leq h_{data}$, lazy update generates less network traffic than eager update if $f_{query} \leq f_{update}$. Of course, there is a price to pay for lazy update: Since the data/query scope is sent to the parent server only when the child server receives a query, the overheads of exchanging the data scope and updates add to the query's response time. In contrast, with eager update, data are refreshed asynchronously with the queries.

6.4 Experiments

The network configuration for our experiments is depicted in Figure 6.9.

To study the benefits of incorporating Data Scoping and Delta Profiling algorithms for disseminating data from the master server to the edge servers, we have conducted a series of experiments using a time-event queuing simulation model. Table 6.2 shows the main parameters used.

The simulation test ground is an isolated system that does not include any external messages that could affect the traffic load. It models the processing, transfer and propagation time of a message (data) by assigning a bandwidth value and processor speed for each link and node respectively, and measuring the delays. The IN/OUT buffer queue length of each node depends on the amount of available IN/OUT link bandwidth and the processor speed. We assume an infinite buffer length for the queues, hence packet drops were not considered in the simulation.

Network Parameters	
Number of User Nodes	12
Node Degree	4
Link Bandwidth (Mbps)	10
Processing Speed at User Node (MBytes/sec)	300
VBTree Parameters	
Number of Data Tuples	10000
VBTree Order	4
Key Size (bytes)	4
Pointer Size (bytes)	4
Digest Size (bytes)	20
Tuple Size (bytes)	500
Query Parameters	
Query Inter-Arrival Time (ms)	10
Number of Tuples in Range Query	10
Spread of Range Queries (%) (in % of Total Number of Data Tuples)	20
Updates Parameters	
Update Inter-Arrival Time (ms)	10
Size of Batch Update	15
Other Parameters	
Simulation Time (ms)	200000

Table 6.2: Cache-Coherence's Experimental Parameters.

6.4.1 Experiment 1: Evaluating Data Scoping and Delta Profiling with Eager and Lazy Updates

In Figure 6.10(a), we see that the query response time for LazyDataScoping is slower than Eager since the edge server triggers off an update with the master server if the server had indicated an update occurance which happened earlier, and all the queries that arrive after this trigger will have to wait for their responses. Whereas in Eager, each update occurrance will immediately push the updated tuples to the edge servers, hence only

queries, that arrive in between the moment the edge server sees the first updated tuple and completely updates its data, will have to wait for their responses. However, if the set of updated data received from the master server is huge, then this updating process might take a long time.

With Data Scoping and having the batch of updates intersecting with this DataScope, Eager is able to improve its response time further as seen in Figure 6.10(b). This is so as the amount of data transferred from the master server to the edge server has been reduced to the amount that is required by the edge server, hence, the transmission time is reduced even with the overhead of transmitting the datascope which is small compared to the massive amount of data transferred from the master server to the edge server without filtering first with the datascope.

From the same Figure 6.10(b), with Delta Profiling, LazyDataScope can improve its response time over EagerDataScopeDeltaProfiling as it can further reduce its amount of data to transfer from the master server to the edge server and at the same time satisfying the query that triggered the update which represents a demand-driven approach as the datascope represents the immediate query scope rather than the previous accumulated query scopes as in EagerDataScope (whose updates may not be required at all at the edge server). In EagerDataScope, when a notice is received, a set of queries will be waiting while the updating process takes place and the transfer is meant for a groups of queries rather than a single query as in Lazy update, hence, the transfer might take longer than if the datascope is meant for a single query. This results in faster individual

query response compared to a response meant for a group of queries whose datascope and result list are larger. However, Lazy update causes larger traffic than eager in general as seen in Figure 6.11 for frequent queries relative to updates as many of these individual updates are required.

6.4.2 Experiment 2: Varying Batch Size and Update Inter-Arrival time

In the first experiment, we vary the batch size from 5 to 55 updates in each batch of updates. With a constant query arrival rate at each edge node and constant update arrival rate at the server, the master server is sending update notifications more frequently with smaller batch size, hence the response time is generally slow for all the methods as nodes need to update their data more frequently, as seen in Figure 6.10. Similarly, more data will have to be tranferred from the master server to the edge servers with more updates.

For the second experiment, we vary the update inter-arrival time from 100 to 1200 ms with each node having a constant query inter-arrival time and the master server with a fixed batch size. This effectively increases the number of updates in each batch. In Figure fig:updatearr, we see that with a smaller update inter-arrival time, the response time for all the update methods increases. This is so as more updates are required for each batch, causing a larger result list to be returned for each update (see Figure 6.12. With a larger result list, the transfer will be slower and the pending queries at the edge

servers will have to wait longer.

6.4.3 Experiment 3: Varying Node Out-Degree

In this experiment, we varying the each node's Out-Degree from 4 to 12. This has the effect of varying the number of hops between the master server and the edge servers. We see from Figure 6.14 and Figure 6.15 that with fewer number of hops (i.e. larger degree), the performance of all the update methods improves. This clearly shows that updates are more efficient if the edge servers are closer to the master server, as it takes a shorter time for data transmission with less number of links to traverse. However, we also observed that when the edge servers are further away from the master server, the update methods perform more effectively when complemented with the Data Scoping and Delta Profiling mechanisms.

6.4.4 Experiment 4: Varying Window Size

In this experiment, we vary the window size at 1, 2, 4 & 8. As seen in Figure 6.16, the response time of EagarDataScopeDeltaProfiling improves as window size increases. This result is expected since with a bigger window size, it is likely that more data updates is carried out for each edge server during the batch update with the master server. Hence, more queries can access the updated values at the edge server but this is at the expense of having a greater amount of data transferred as seen in Figure 6.17. As for the Lazy

updates, they are insensitive to window size changes as each edge server data refresh is meant only for one query (on-demand) and hence no windowing is implemented.

6.4.5 Experiment 5: Varying Tuple Size

In this experiment, we vary the data tuple size from 300 to 1200 bytes. As expected, with the size of each tuple increases, the result list will also be larger, hence, the response time for all the methods increases with increasing tuple size as seen in Figure 6.18.

6.4.6 Experiment 6: Varying Link Bandwidth

In this experiment, we vary the link bandwidth from 1 to 10 Mbps. As in any data transfer through a network, with an increasing bandwidth capacity, more data can be transferred at a shorter time. From Figure 6.19, it clearly shows that the performance of all the methods improve.

Hence, through this series of experiments, we see that regardless of the numerous parameters we vary, the Data Scope and Delta Profiling mechanisms clearly helps in improving the performance of data updates through a network.

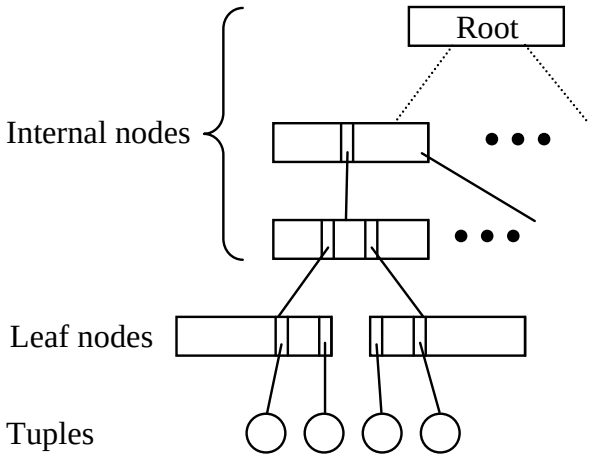
6.5 Summary

This chapter introduces an efficient data dissemination solution to improve the scalability of edge computing, by minimizing or even eliminating redundant updates to the edge

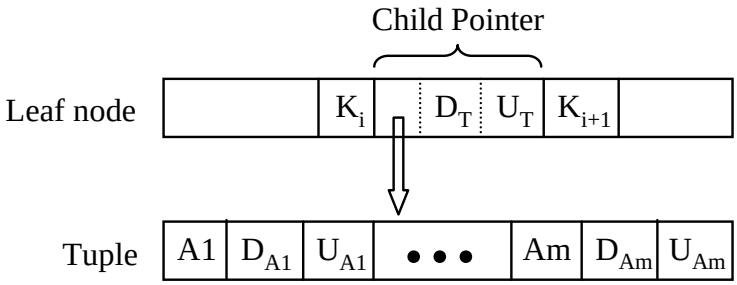
servers. Our key contributions include two novel mechanisms – data scoping and delta profiling.

Data scoping automatically monitors the active tuples and attributes that are targeted by the queries at each edge server, and uses this information to demarcate the local data set that should be maintained at that edge server. Those data values within the data scope that are out-of-date and need to be refreshed are then picked up by the *delta profiling* mechanism in logarithmic time (except when a substantial fraction of the key values in the table has been changed), all without requiring the master server to keep track of the data versions at any edge server.

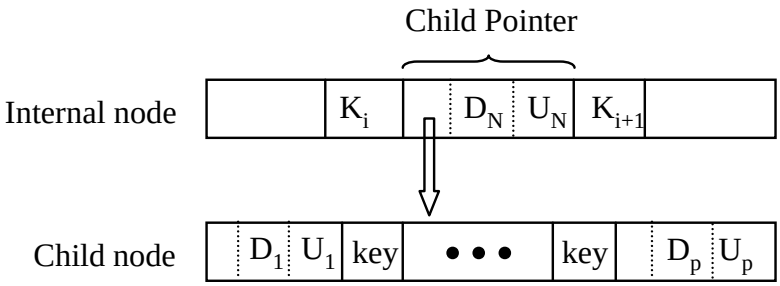
Analysis and experiment studies confirm that: (1) With eager update, data scoping can be very beneficial if a substantial number of tuples and/or attributes are not targeted by the queries at an edge server. (2) Delta profiling is a very effective supplement to data scoping when lazy update is employed. (3) Lazy update reduces network traffic relative to eager update when queries are less frequent than updates, at the expense of potentially large deterioration in query response time.



(a) Overview



(b) Structure of Leaf Node



(c) Structure of Internal Node

Figure 6.2: Verifiable B-Tree

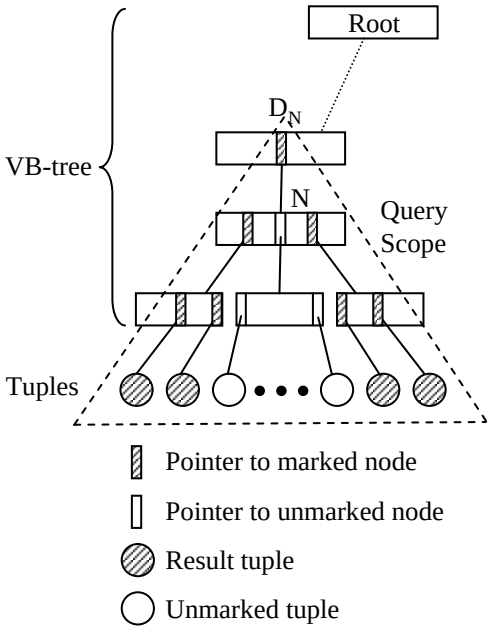


Figure 6.3: Query Scope

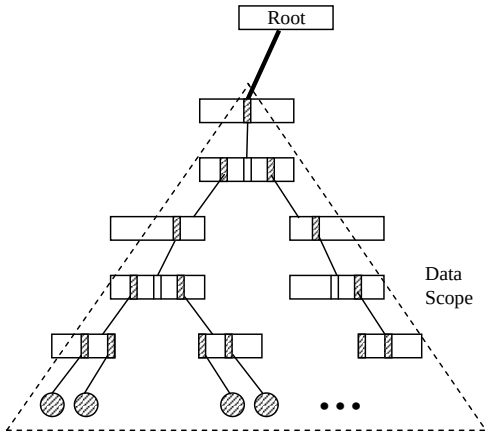


Figure 6.4: Data Scope

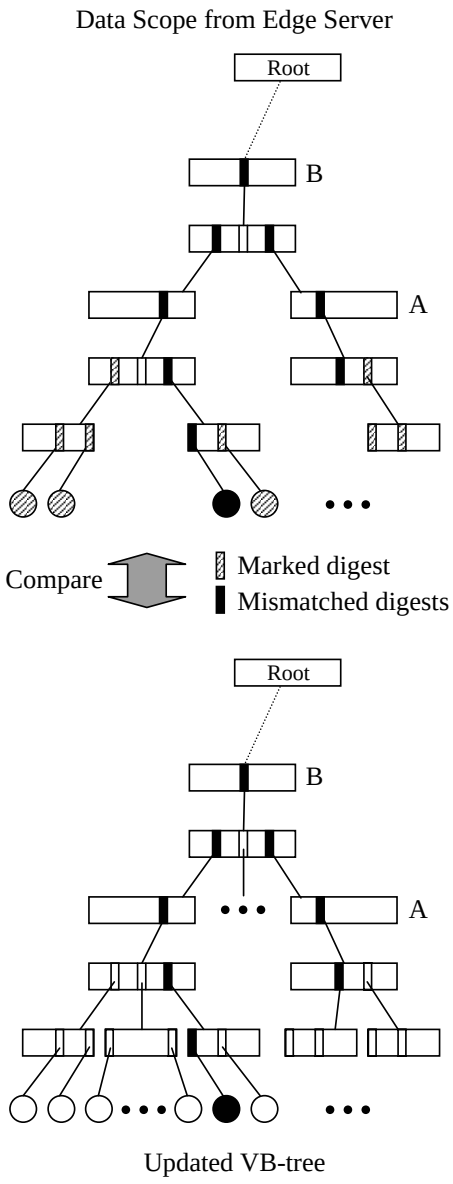
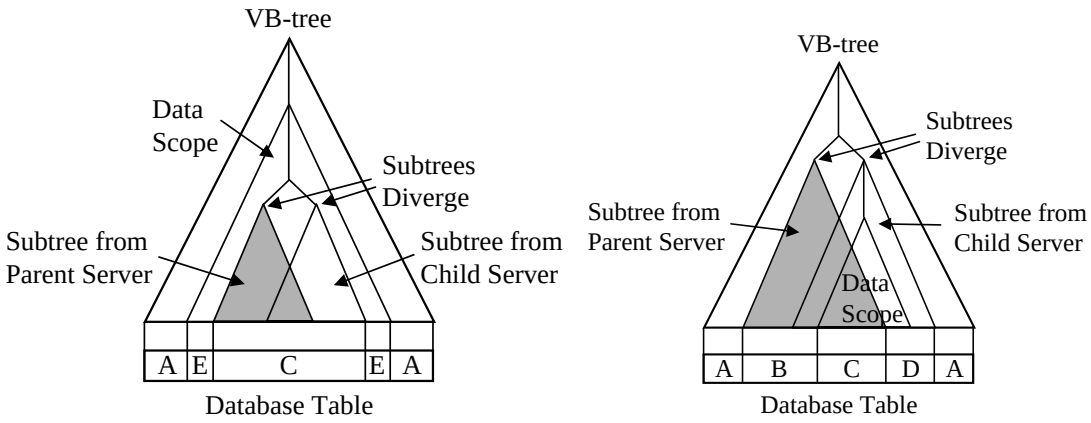


Figure 6.5: Delta Profiling



(a) Divergence Within Data Scope (b) Divergence Above Data Scope

Figure 6.6: Changes to VB-tree Structure

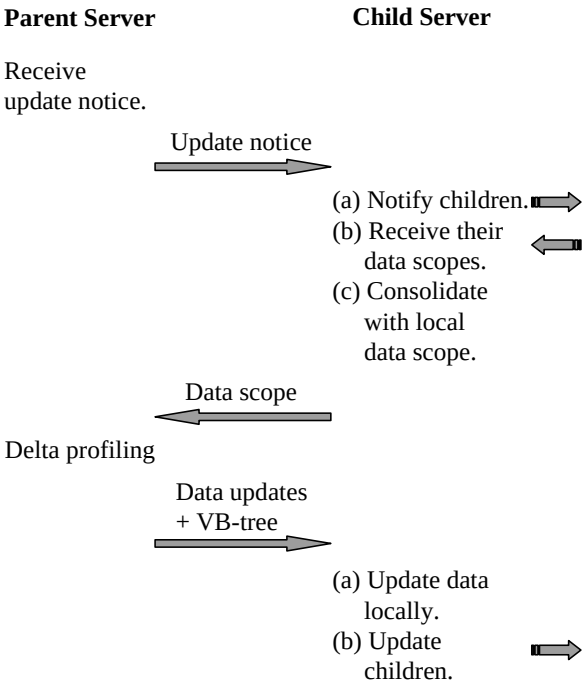


Figure 6.7: Protocol for Eager Update

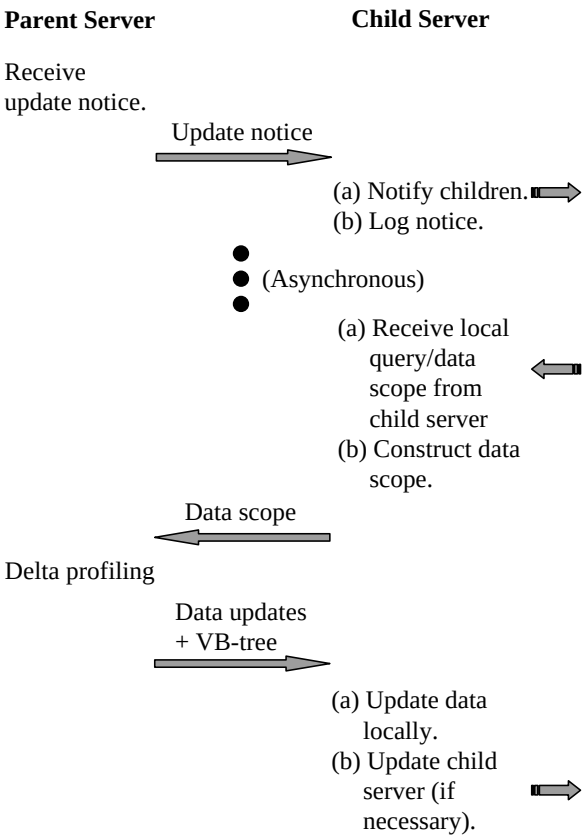


Figure 6.8: Protocol for Lazy Update

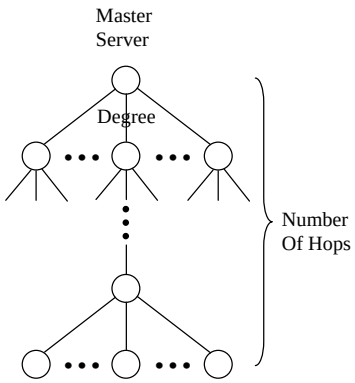
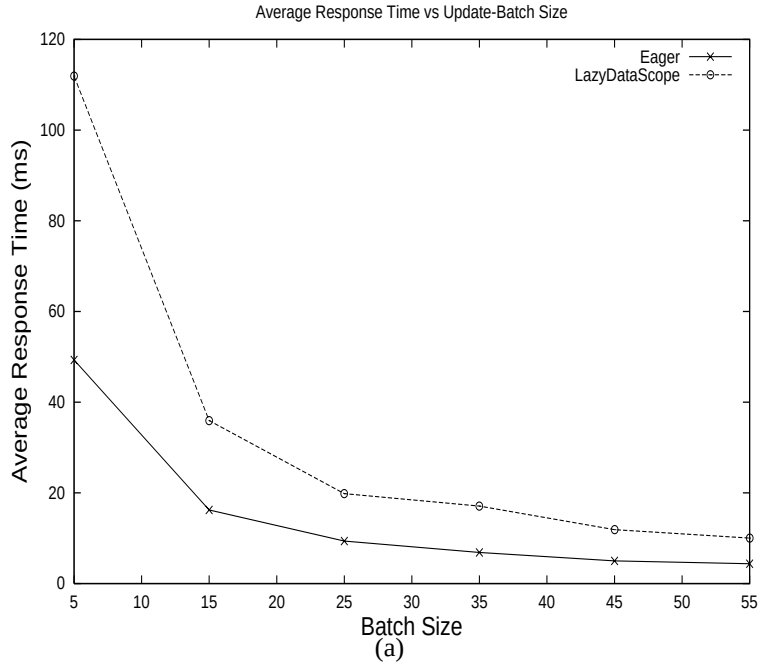
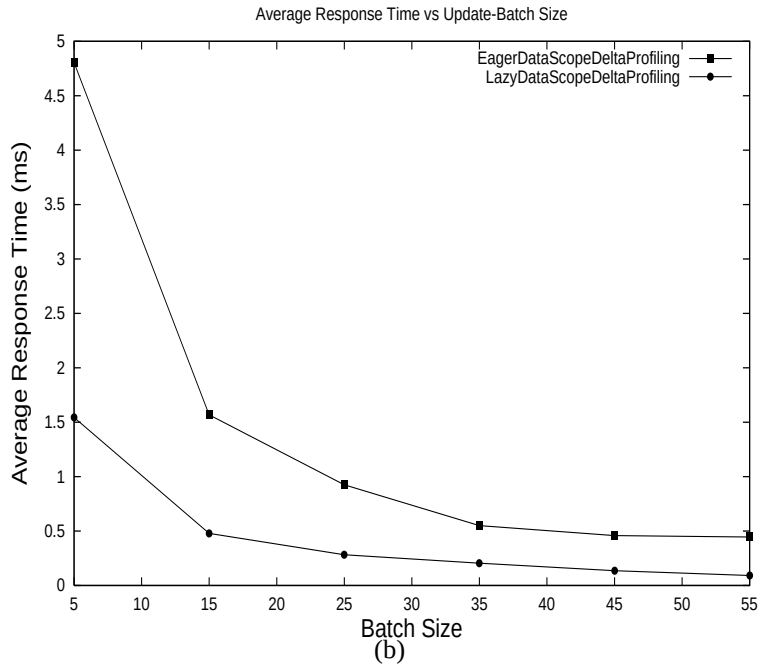


Figure 6.9: Network Configuration



(a)



(b)

Figure 6.10: Batch Size, Response Time.

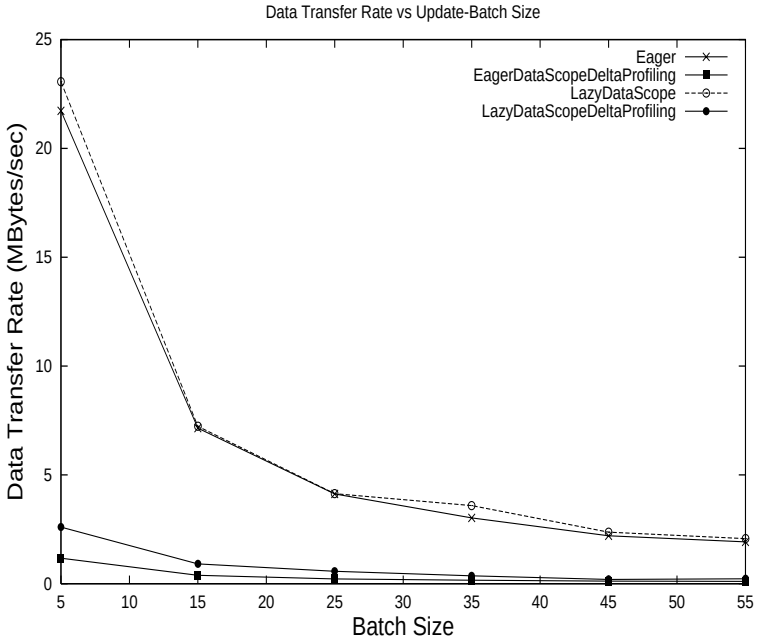


Figure 6.11: Batch Size, Data Rate.

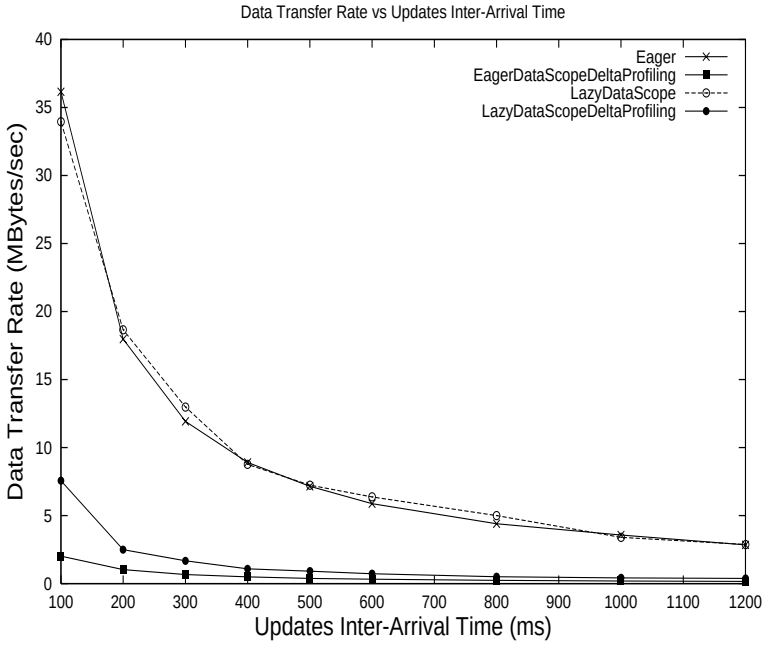
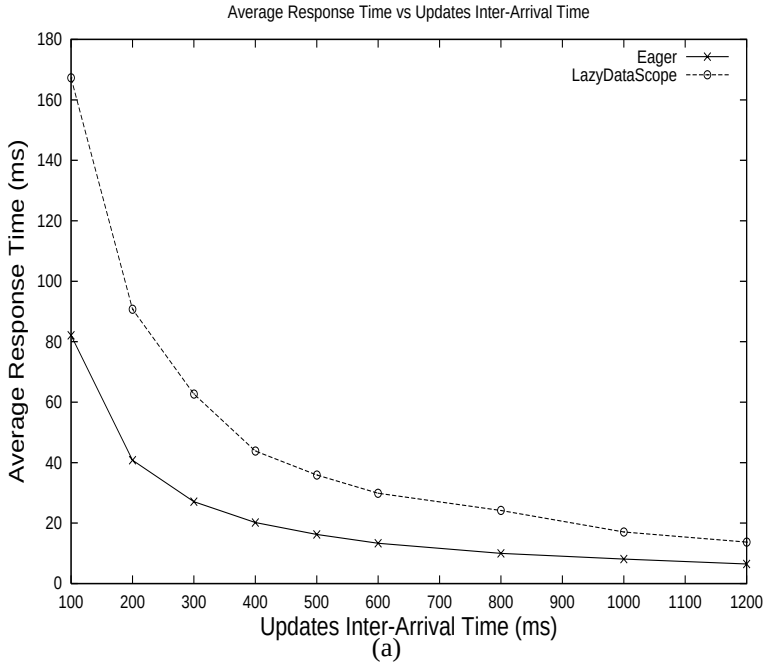
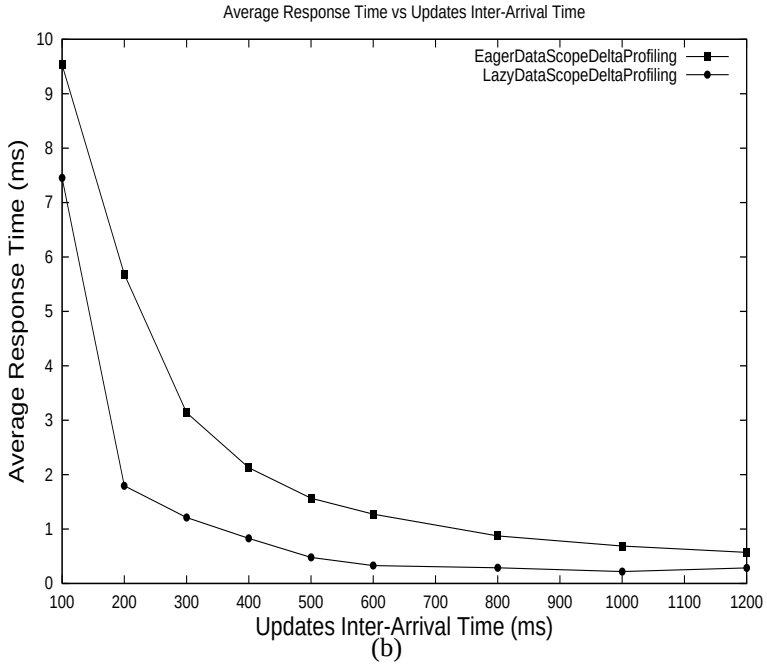


Figure 6.12: Update Inter-Arrival Time, Data Rate.

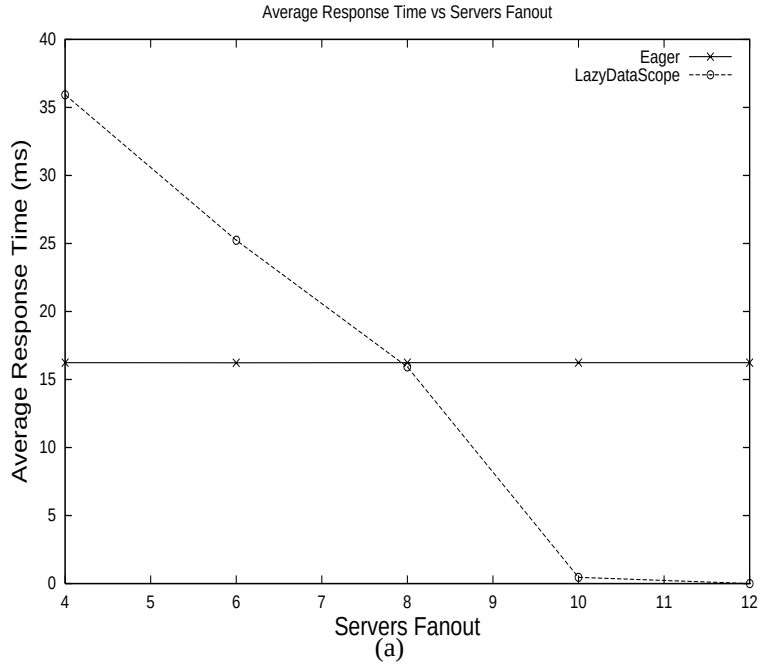


(a)

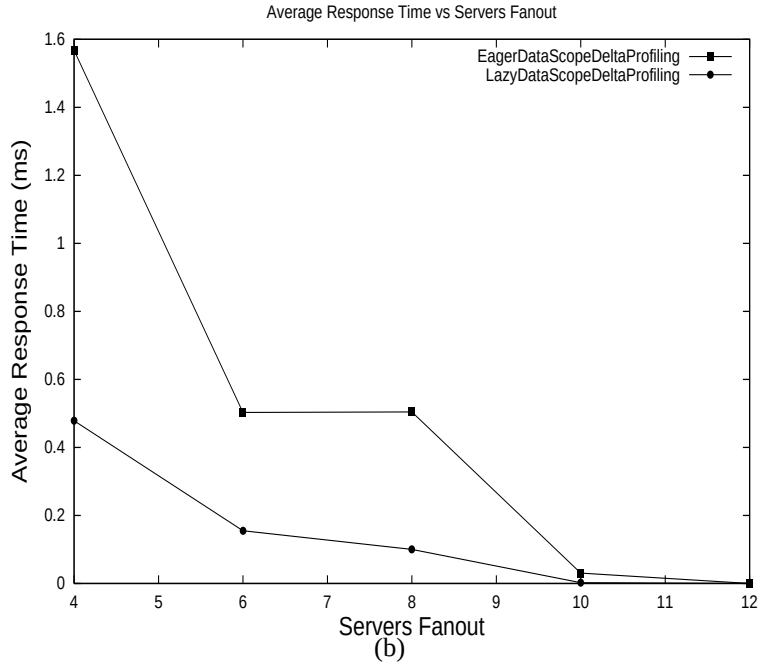


(b)

Figure 6.13: Update Inter-Arrival Time, Response Time.



(a)



(b)

Figure 6.14: Node Fanout, Response Time.

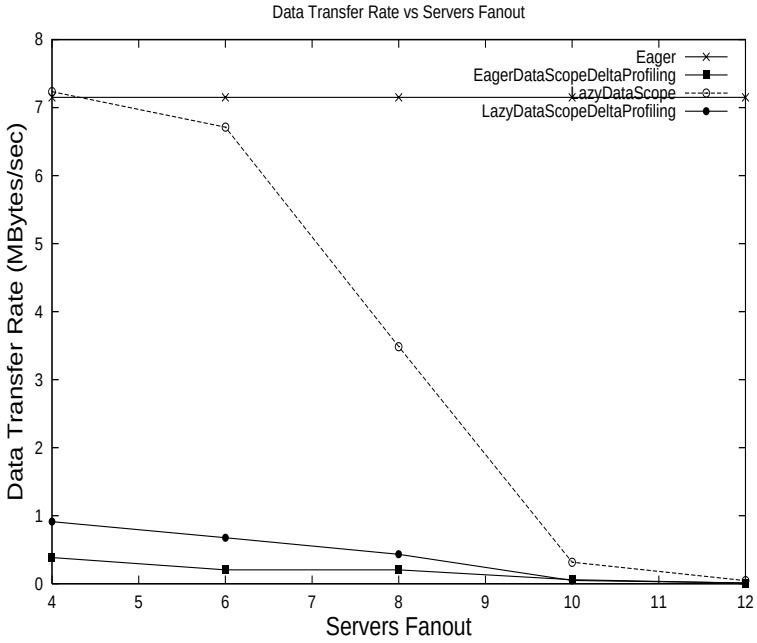


Figure 6.15: Node Fanout, Data Rate.

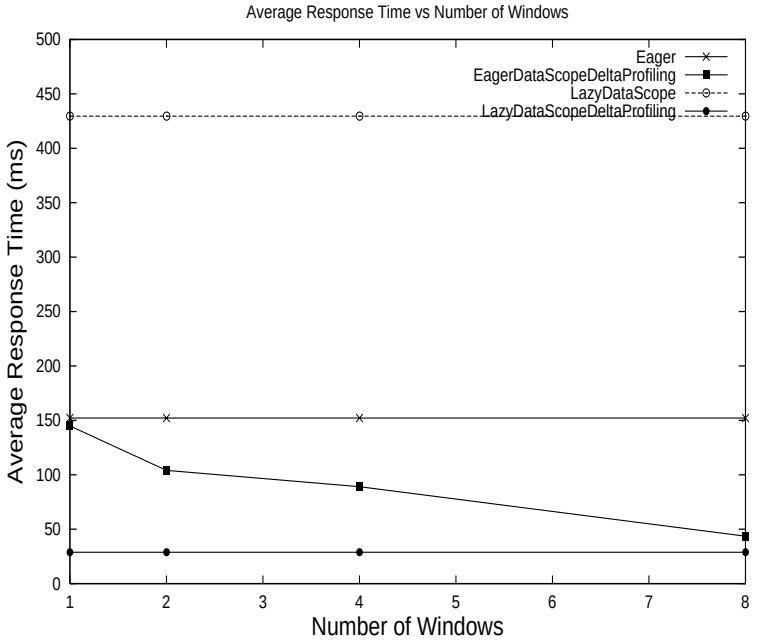
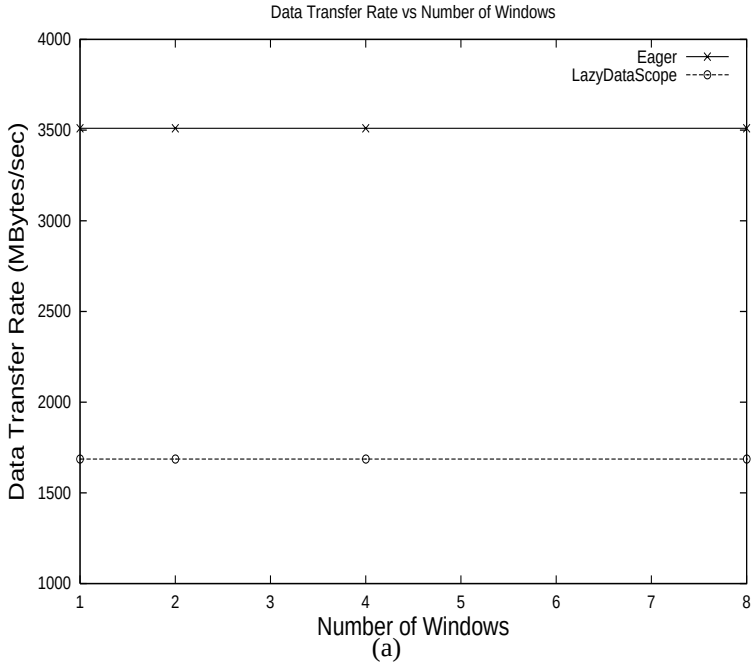
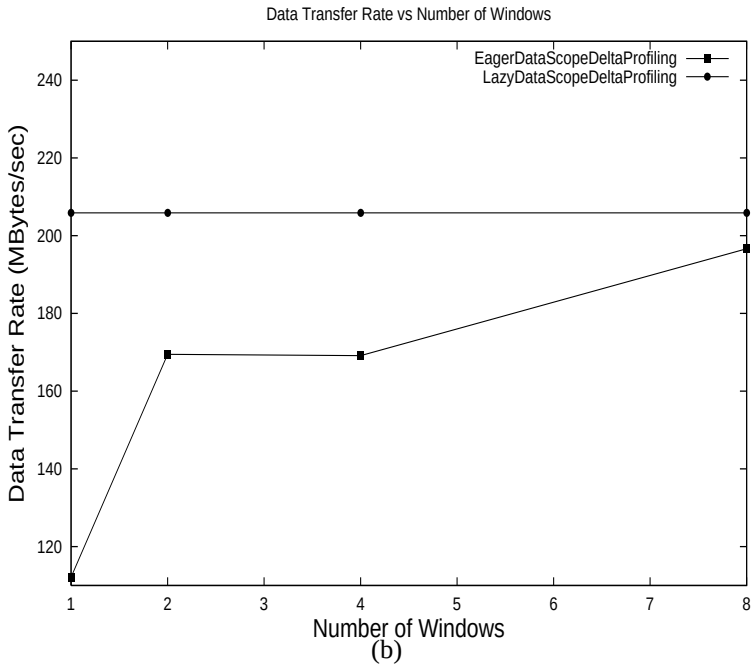


Figure 6.16: Window Size, Response Time.

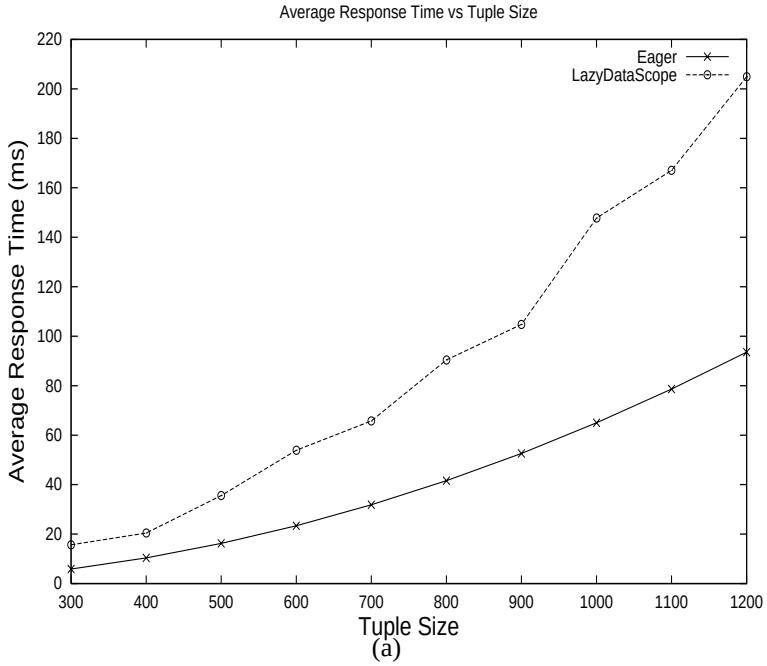


(a)

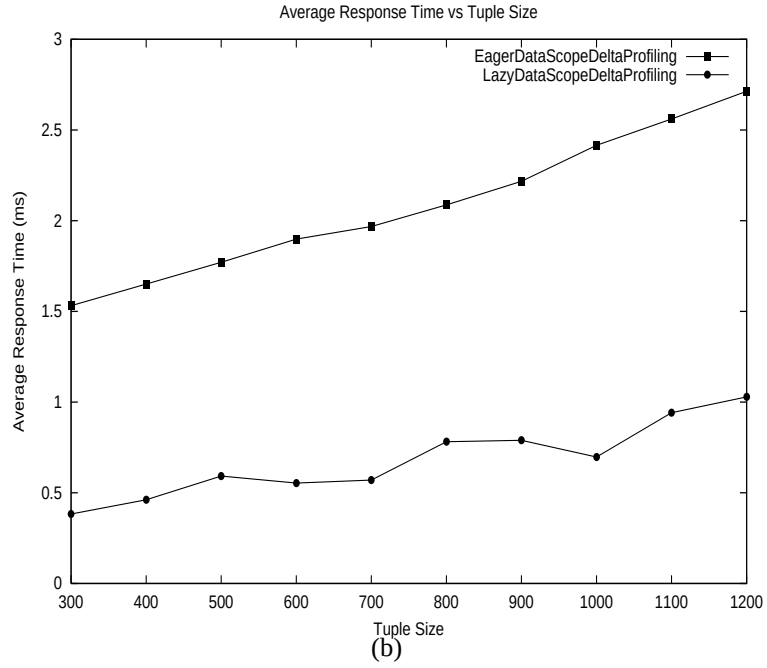


(b)

Figure 6.17: Window Size, Data Rate.

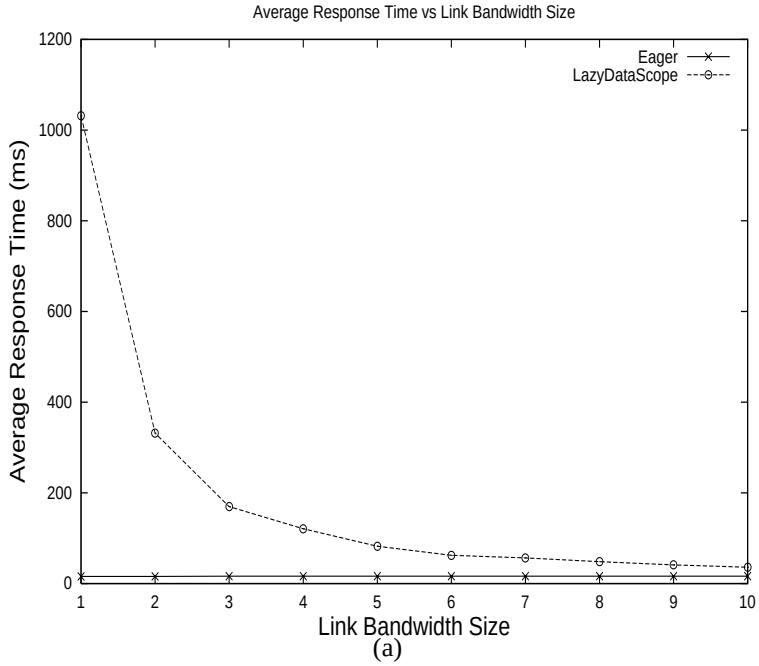


(a)

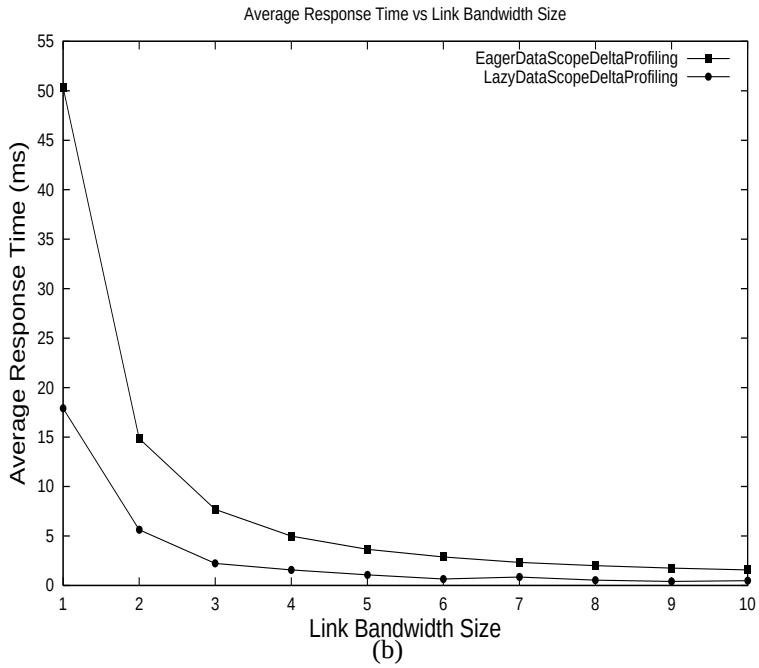


(b)

Figure 6.18: Tuple Size, Response Time.



(a)



(b)

Figure 6.19: Link Bandwidth, Response Time.

Chapter 7

Conclusion

7.1 Conclusion

Storage resources can be managed in different ways especially when the stored content can be accessed or manipulated at the hosts. We have identified several problems in managing and materializing data in caches for answering queries both in centralized and distributed environments. In this thesis, we have proposed several algorithms and techniques to manage the caches to improve the performance in query processing.

In the centralized settings, we have proposed a novel demand-driven caching framework, called *cache-on-demand* (CoD). CoD views intermediate/final answers of existing running queries as *virtual* caches that an incoming query can exploit. Those caches that are beneficial may then be materialized for the incoming query. Such an approach is essentially non-speculative: the exact cost of investment and the return on investment

are known, and the cache is certain to be reused! We addressed several issues for CoD to be realized. We also propose three optimizing strategies: Conform-CoD, Scramble-CoD and Integrated-CoD. Conform-CoD and Scramble-CoD are based on a two-phase optimization framework, while Integrated-CoD operates in a single-phase framework. We conducted extensive performance study to evaluate the effectiveness of these algorithms. Our results show that all the CoD-based schemes can provide substantial performance improvement when compared with a predictive scheme and a no-caching scheme. Moreover, we show that Integrated-CoD offers the best performance but incurs the highest optimization overhead. Conform-CoD, which performs the worst in most cases, has the least optimization overhead. In addition, we have included several CoD extensions, to improve the overall performance of the query evaluation engine. It integrates three new techniques to realize this performance gain. The first method exploits intra-query parallelism where a sequence of operators within a query execution plan are executed in a pipeline. The second method explores the advantage of keeping multiple plans to increase the match space of CoD virtual caches at the expense of memory and comparison overhead. Lastly, the execution orders of plans may be reordered by the plan scheduler to further promote cache reuse.

Secondly, we propose CacheWire to instill collaboration among caches in a Peer-to-Peer(P2P) environment. We based our work on two observations on human behavior. First, when we need information, we usually ask our friends around us. Interestingly, our friends usually remember what we ask, and will come back to us if they need the

same information later (knowing that we may have had the information since we have previously asked for them). Second, whenever we want to discard an item that is still usable (perhaps because we are clearing our office, moving, or have no need of the item anymore), we usually would pass it to a friend (or even charity organization) who have need of it. This prompted us to introduce the idea of a peer checking with its neighbors before it trashed out any cached objects. This is particularly beneficial if the cached objects are computationally expensive to produce or the communication overhead may be high.

This architecture can be introduced to promote collaboration between network hosts in capitalizing their local knowledge to share their resources for answering its own as well as other URL/Query requests. It setup a framework that wires up available and willing network host/proxy caches qualitatively and constructively which allows hosts to become adaptive and community friendly. It is able to adjust its resources to keep items that are useful and at the same time share what it has learned with others. CacheWire supports decentralized collaboration between web proxy servers or application clients in a P2P environment. It learns, processes and remembers as it listens, and makes decisions that are based on the acquired knowledge. When a purpose arises, it handles it with the information at hand and makes constructive communication with others whenever necessary. Six different CacheWire's options have been evaluated for both Web Objects and OLAP queries. Our results showed that CacheWire's components with selective collaboration generally contribute to a higher detailed cost saving ratio.

Lastly, we propose two mechanisms for efficiently maintaining cache coherency. The first mechanism demarcates the local data set at each edge server, and another mechanism that identifies in logarithmic time those updates that apply to the local data set. The net result is that only updates to those portions of the database that are required by individual edge servers are propagated to satisfy their users and applications. The mechanisms work in conjunction with both eager and lazy update models. Analysis and experiment studies confirm that the proposed mechanisms can be very effective in minimizing redundant updates to the edge servers.

7.2 Future Research Directions

In the cache-on-demand mechanism proposed in the centralized environment, we have restricted our work to multi-join queries, generalizing to other types of queries is certainly a possible extension. Another possible work can include a study on how to adaptively determine an optimal value for the number of relations to be involved in the virtual cache.

In the distributed environment, the advances in the networking arena has prompted the feasibility of database connectivity across the Internet. Furthermore, the increased availability of computing resources and improved networking infrastructure have allowed the ease of data exchange between any two distant points. Hence, collaboration and sharing of data and information in distributed database systems can be very feasible

and become very beneficial in future. An interesting further work can involve an empirical or detailed study of the co-relation between real users of variable domains to identify the similarity between the query access patterns.

Also, in addition to the proposed cache coherency mechanism, it will be interesting to investigate how the edge servers in the dissemination tree can be re-organized dynamically according to their respective data scopes. Another avenue for further research is to combine our solution with the coherence-based propagation in [84]. Further, it will be insightful to conduct an investigation on the performance of the proposed techniques in a real environment.

All in all, caches often exist in systems to provide that extra scalability, reliability and performance. Hence, we believe that the work to exploit the use of caches in this field holds huge potential for future research and development.

Bibliography

[1] Secure hash standard (shs). *National Institute of Standards and Technology, FIPS Publication*, 180-1, April 1995.

[2] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 169–180. Morgan Kaufmann Publishers Inc., 2001.

[3] Akamai. www.akamai.com.

[4] M. Altnel, C. Bornhvd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 718–729, 2003.

[5] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, and T. Zhong. Web caching for database applications with oracle web cache. In *Proceedings of the ACM*

SIGMOD international conference on Management of data, pages 594–599. ACM Press, 2002.

[6] G. Barish and K. Obraczka. World wide web caching: Trends and techniques. *IEEE Communications Magazine, Internet Technology Series*, 38(5):178–185, May 2000.

[7] M. Bjornsson and L. Shrira. Buddycache: High-performance object storage for collaborative strong-consistency applications in a wan. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming*, pages 26–39, 2002.

[8] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 97–108. ACM Press, 1999.

[9] Y. Breitbart and H. Korth. Replication and consistency: being lazy helps sometimes. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 173–184. ACM Press, 1997.

[10] L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the web. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 550–561, 2002.

[11] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: the devil is in the details. *SIGMETRICS Perform. Eval. Rev.*, 26(3):11–15, 1998.

[12] Squid Web Proxy Cache. www.squid-cache.org.

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

[14] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Conference of the IEEE Computer and Communications Societies (INFOCOMM)*, pages 294–303, March 1999.

[15] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *In Proceedings of the USENIX Annual Technical Conference*, 1996.

[16] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th International Conference on Data Engineering*, pages 190–200, April 1995.

[17] C. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In

Proceedings of the International Conference on Extending Data Base Technology, pages 323–336, March 1994.

[18] M. Chen, M. Lo, P. Yu, and H. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 15–26, August 1992.

[19] S. Chen, P. Gibbons, T. Mowry, and G. Valentin. Fractal prefetching b+-trees: optimizing both cache and disk performance. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 157–168. ACM Press, 2002.

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

[21] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Lecture Notes in Computer Science*, pages 46–67, 2000.

[22] E. Cohen and H. Kaplan. Exploiting regularities in web traffic patterns for cache replacement. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 109–118. ACM Press, 1999.

[23] B. Cooper and H. Molina. Peer-to-peer data trading to preserve information. *ACM Trans. Inf. Syst.*, 20(2):133–170, 2002.

[24] A. Crespo and H. Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 23–33. IEEE Computer Society, 2002.

[25] N. Dalvi, S. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 59–70, Santa Barbara, CA, May 2001.

[26] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341, Mumbai, India, August 1996.

[27] P. Deshpande and J. Naughton. Aggregate aware caching for multi-dimensional queries. In *Proceedings of the International Conference on Extending Data Base Technology*, pages 167–182, Konstanz, Germany, March 2000.

[28] P. Deshpande, K. Ramasamy, A. Shukla, and J. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 259–270, Seattle, Washington, June 1998.

[29] Cisco Cache Engine. www.cisco.com/go/cache.

[30] A. Evans, W. Kantrowitz, and E. Weiss. A user authentication system not requiring secrecy in the computer. In *Communications of the ACM*, volume 17(8), pages 437–442, August 1974.

[31] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 251–262. ACM Press, 1999.

[32] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. volume 28, pages 254–265. ACM Press, 1998.

[33] M. Freedman and D. Mazieres. Sloppy hashing and self-organizing clusters. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[34] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large internet caches. In *In Proceedings of The Sixth Workshop on Hot Topics in Operating Systems*, 1997.

[35] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proceedings of the twelfth international conference on World Wide Web*, pages 449–460. ACM Press, 2003.

[36] Gnutella. www.gnutella.com.

[37] S. Goh, B. Ooi, and K. Tan. An efficient method for queries execution in a multi-user environment. In *Proceedings of the 7th International Symposium on Database Systems for Advanced Applications*, pages 312–319, Hong Kong, PRC, April 2001.

[38] S. Goh, B. Ooi, and K. Tan. Demand-driven caching in multiuser environment. In *IEEE Transactions on Knowledge and Data Engineering*, pages 112–124, January 2004.

[39] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.

[40] R. Hankins and J. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 1–12, 2003.

[41] Hyper Text Caching Protocol HTCP. Hyper text cache protocol. www.htcp.org.

[42] Internet Caching Protocol ICP. Internet cache protocol. icp.ircache.net.

[43] Icq. www.icq.com.

[44] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 312–321, Atlantic City, NJ, May 1990.

[45] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 213–222. ACM Press, 2002.

[46] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.

[47] P. Kalnis, W. Ng, B. Ooi, D. Papadias, and K. Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 25–36. ACM Press, 2002.

[48] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM Press, 1997.

[49] Kazaa. www.kazaa.com.

[50] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[51] D. Kossmann, M. Franklin, G. Drasch, and W. Ag. Cache investment: Integrating query optimization and dynamic data placement. *ACM Transactions on Database Systems*, 25(4):517–558, 2000.

[52] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 371–382. ACM Press, 1999.

[53] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 391–400. Morgan Kaufmann Publishers Inc., 2001.

[54] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, 1992.

[55] B. Lan, S. Bressan, B. Ooi, and K. Tan. Rule-assisted prefetching in web-server caching. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 504–511. ACM Press, 2000.

[56] P. Larson, J. Goldstein, and J. Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *Proceedings of the 20th International Conference on Data Engineering*, pages 177–189. IEEE Computer Society, 2004.

[57] W. Lou and H. Lu. Efficient prediction of web accesses on a proxy server. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 169–176. ACM Press, 2002.

[58] T. Loukopoulos, P. Kalnis, I. Ahmad, and D. Papadias. Active caching of on-line-analytical-processing queries in www proxies. In *Proceedings of the International Conference on Parallel Processing*, pages 419–426. IEEE Computer Society, 2001.

[59] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002.

[60] D. Margulius. Apps on the edge. *InfoWorld*, 24(21), May 2002. http://www.infoworld.com/article/02/05/23/ 020527feedgetci_1.html.

[61] H. Matthew, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 242–259. Springer-Verlag, 2002.

[62] M. Mitzenmacher. Compressed bloom filters. In *PODC: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. ACM Press, 2001.

[63] C. Mohan. Caching technologies for web applications. In *Proceedings of the 27th International Conference on Very Large Data Bases*, page 726. Morgan Kaufmann Publishers Inc., 2001.

[64] Napster. opennap.sourceforge.net.

[65] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 73–84. ACM Press, 2002.

[66] OpenCola. www.openp2p.com/pub/a/p2p/2001/05/24/oram.html.

[67] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *The VLDB Journal*, 8(3-4):305–318, 2000.

[68] H. Pang and K. Tan. Authenticating query results in edge computing. In *Proceedings of the 20th International Conference on Data Engineering*, pages 560–572. IEEE Computer Society, 2004.

[69] Microsoft Proxy. www.microsoft.com/isaserver.

[70] S. Ratnasamy, P. Francis, M. Handley, and R. Karp. A scalable content-addressable network. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[71] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for dhts: Some open questions. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 45–52. Springer-Verlag, 2002.

[72] Relais. www-sor.inria.fr/projects/relais/.

[73] R. Rivest. Rfc 1321: The md5 message-digest algorithm. *Internet Activities Board*, April 1992.

[74] P. Rodriguez, C. Spanner, and E. W. Biersack. Analysis of web caching architectures: hierarchical and distributed caching. *IEEE/ACM Trans. Netw.*, 2001.

[75] A. Rosenthal and U. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 230–239, Los Angeles, CA, August 1988.

[76] A. Rousskov and D. Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22-23):2155–2168, 1998.

[77] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001.

[78] P. Roy, K. Ramamritham, S. Seshadri, P. Shenoy, and S. Sudarshan. Don't trash your intermediate results, cache 'em. In *CoRR (Number 0003005)*, March 2000.

[79] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi-query optimization. In *Proceedings of the ACM-SIGMOD In-*

ternational Conference on Management of Data, pages 249–260, Dallas, Texas, June 2000.

[80] P. Sarkar and J. Hartman. Hint-based cooperative caching. *ACM Trans. Comput. Syst.*, 18(4):387–419, 2000.

[81] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 51–62, Mumbai, India, August 1996.

[82] T. Sellis. Multiple query optimization. *ACM Transactions on Databases*, 13(1):23–52, March 1988.

[83] S. Shah, S. Dharmarajan, and K. Ramamritham. An efficient and resilient approach to filtering and disseminating dynamic data. In *Proceedings of the 29th Conference on Very Large Data Bases*, pages 57–68, September 2003.

[84] S. Shah, K. Ramamritham, and P. Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *Proceedings of the 28th Conference on Very Large Data Bases*, pages 526–537, August 2002.

[85] E. Shekita, H. Young, and K. Tan. Multi-join optimization for symmetric multiprocessors. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 479–492, August 1993.

[86] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic caching of query results for decision support systems. In *Proceedings of the International Conference on Scientific and Statistical Databases*, pages 254–263, Cleveland, Ohio, July 1999.

[87] R. Srikant and Y. Yang. Mining web logs to improve website organization. In *WWW: Proceedings of the tenth international conference on World Wide Web*, pages 430–437, New York, NY, USA, 2001. ACM Press.

[88] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. volume 11, pages 17–32. ACM Press, 2003.

[89] S. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient views. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 319–330, Seattle, WA, June 1998.

[90] K. Tan, S. Goh, and B. Ooi. Cache-on-demand: Recycling with certainty. In *Proceedings of the 17th International Conference on Data Engineering*, pages 633–640, Heidelberg, Germany, April 2001. IEEE Computer Society.

[91] K. Tan and H. Lu. Workload scheduling of multi-join queries. *Information Processing Letters*, 55(5):251–257, 1995.

[92] A. Tomasic and H. Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 129–138. ACM Press, 1993.

[93] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. In *Internet Draft*, 1998.

[94] J. Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, 25(9):36–46, October 1999.

[95] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 206–215, October 2000.

[96] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 188–199. Morgan Kaufmann Publishers Inc., 2000.

[97] J. Yang, W. Wang, and R. Muntz. Collaborative web caching based on proxy affinities. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 78–89. ACM Press, 2000.

[98] O. R. Zaane, M. Xin, and J. Han. Discovering web access patterns and trends by applying olap and data mining technology on web logs. In *ADL: Proceedings of*

the Advances in Digital Libraries Conference, page 19, Washington, DC, USA, 1998. IEEE Computer Society.

[99] M. J. Zaki. Efficiently mining frequent trees in a forest. In *KDD: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, New York, NY, USA, 2002. ACM Press.

[100] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UC Berkeley 2001.