Using Clustering for Effective Management of a Semantic Cache in Mobile Computing

Qun Ren Margaret H. Dunham
Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275
qren, mhd@seas.smu.edu

Abstract

Semantic caching is particularly attractive for use in a mobile computing environment due to its content-based reasoning ability and the fact that only the required data (as opposed to a file or pages of data) is transmitted over the wireless link. In this paper we propose a cluster based approach to manage a semantic cache, aiming at examining and further making use of the semantic locality among queries in a workload. Cache replacement performance using this technique is shown to outperform that of an LRU scheme where only temporal locality is considered.

1 Introduction

In a mobile computing environment, mobile clients and database servers communicate via wireless channels that have limited bandwidth and are vulnerable to disconnection ([IK96]). Caching of frequently accessed data at the mobile client side can reduce the interactions between the client and server, and therefore is an effective way in handling these problems ([CSL98]). Traditional cache schemes, such as page and tuple caching, do not keep the semantic information about cached data and thus it is impossible to support content-based reasoning. The idea of *semantic caching* is that the mobile client maintains in the cache both the semantic descriptions and associated answers of previous queries ([DFJ96], [KB96]). When a new query comes, the mobile client can determine whether it can be totally answered, how much it can be answered, and what data are missing. This ability greatly reduces the wireless traffic because only the needed data are transferred. It helps with disconnection too: total or partial results may be obtained even when the server is unreachable.

Instead of exploiting static spatial and temporal locality, semantic caching makes use of *semantic locality* among queries ([DFJ96]). The performance of a semantic cache depends on how much the queries are *semantically related* with each other. It also depends on how effectively the cache is managed, that is to say, whether the cache management strategies can find and utilize the semantic locality. In this paper, we propose a *clustering* approach which clusters together the semantically related query results in the cache. Thus if a query Q_2 can be totally or partially answered by Q_1, it is put in the same cluster as Q_1. We then build our cache replacement strategy, *Two-Level LRU*, on the clustering structure exploiting both semantic and temporal locality. The performance of Two-Level LRU in mobile computing is validated through a simulation study, which shows its effectiveness in reducing wireless network traffic and dealing with disconnection.

The rest of this paper is organized as follows. In Section 2, we review previous work on semantic caching. Section 3 first introduces a basic semantic cache model, then presents the idea of clustering, Two-Level LRU is also examined there. Performance study results are discussed in Section 4. We summarize our work and discuss future research in Section 5.

2 Related Work

Semantic caching has been widely studied in centralized systems ([Rou91], [CR94]), client-server environment ([DFJ96], [KB96]), OLAP systems ([DRSN98]), mobile computing ([CTO97]) and heterogeneous systems ([GG97]). In this section, we briefly review previous semantic caching work, which mainly focus on cache replacement strategies. Other issues, such as cache coherence control examined in [CTO97], are not considered in this paper.

A profit-based or benefit-based replacement scheme is proposed in [KB96]. Every cached predicate is assigned a benefit metric, which takes into account its result size, frequency of reference, and execution cost etc. The one with the smallest benefit value will then be chosen as the replacement victim. The strategy they use is very similar to that of [SSV96]. [DRSN98] works in the OLAP domain, cache replacement is done at the *chunk* level and uses a benefit-based CLOCK policy. The benefit metric mainly involves execution cost which varies from chunk to chunk in OLAP applications. Unlike these works, [DFJ96] adopts several other cache replacement schemes. In addition to the traditional *LRU/MRU*, [DFJ96] further utilizes the semantic locality in semantic caching replacement. For each cached *semantic region*, replacement value is assigned which is the negative of the *Manhattan distance* between the "center of gravity" of that region and the "center of gravity" of the most recent query. With this distance function, semantic regions that are semantically "closer" to the current query are less likely to be discarded. The rationale used here is to predict the future use patterns for the segments by examining their semantic relationships with the current query.

We also utilize semantic information to do replacement in this research. However, unlike [DFJ96] who discards data according to the criteria of "Manhattan distance", which is difficult and costly to compute in most cases, we use a more straightforward clustering approach for cache replacement.

3 Clustering and Semantic Cache Management

In this section, we first describe a basic semantic cache model which provides the base for this research ([RD98]). We then propose the clustering approach which aims at examing semantic locality among queries, and based on that, we further give the replacement strategy - Two-Level LRU.

3.1 Basic Semantic Cache Model

We discuss the semantic cache model from the aspects of definition, organization and operation. A *semantic cache* is defined to be composed of a set of distinct *semantic segments*, which are original, decomposed or coalesced query results. A semantic segment S can always be specified by the tuple $< S_R, S_A, S_P, S_C >$, where S_R and S_A define the base relation and attributes that are involved in the creation of S, S_P indicates the criteria that the tuples in S satisfy, and S_C represents the actual content of S. Physically, we group tuples in S_C into pages. Hence every segment is stored in one or more linked pages of the same fixed size(For a detailed discussion see [RD98]).

A semantic cache is managed at a semantic segment level, and we organize it by maintaining an index which contains the semantic as well as the physical storage information about the cached segments. In what follows, we list the basic items kept in the index which are related with cache replacement. Other information may need to be added when more issues such as coherency control are examined.

- the name S, the base relation S_R, the attribute set S_A, the selection predicate S_P;
- the pointer pointing to the first page that stores the segment;
- the timestamp indicating when the segment was last accessed S_{TS};

When a query Q is submitted on the client side, it is first processed locally via the semantic cache. The candidate segments that contribute to Q's result are determined from the cache index. If Q can be partially answered by a semantic segment S, it is split into two disjoint parts: a *probe* query which retrieves the portion of the result available in S, and a *remainder* query which defines any other missing tuples of Q. This procedure is called *query trimming*. The remainder query of Q will then be trimmed by the next candidate segment. This process continues until there is no candidate segment in the cache or the remainder query becomes empty. If the final remainder query is not empty, it is sent to the database server in the fixed network for processing. At last, the result of Q is obtained by coalescing every partial result([RD98]).

Three disjoint parts are left after a query Q is partially answered by a segment S: the intersection of Q and S, the difference of S with respect to Q, and the difference of Q with respect to S. To better reflect the frequency of reference, we coalesce Q's parts together, cache it as a new segment, and mark it with the current timestamp. Meanwhile, S gets decomposed, the portion of S not involved in answering the query becomes another segment and keeps the original timestamp of S.

3.2 Clustering

Semantic caching is built on the assumption of semantic locality, which is the property that if a query has been requested, other queries semantically related to it are also likely to be requested in the near future. Here, "semantically related" means the queries have resulting tuples in common. Examing the *semantic relationship(SR)* between queries can help locate semantic locality among queries and thus facilitate cache management. For example, if space is limited we won't admit a query to the cache if it is not semantically related with others, because it may not be within the user's hot interest region and thus less likely to be reused later. For a similar reason, those cached queries which are less related with others have a higher possibility to be discarded from the cache. Although SR can be easily defined, such as the ratio of the overlapped result size to the size of the total query results, it can not be easily determined before the queries are processed, especially when their predicates involve different attribute sets. Thus in our work we use clustering as an approach to approximate the true SR value.

Clustering is proposed as a straightforward and effective way to investigate semantic relationships and explore the semantic locality. The idea is to divide queries into groups, with those semantically related or *adjacent* being put together, and call each group a *cluster*. Here, if two queries have portions involved in the third query, we call them semantically adjacent with each other. Clustering queries in a workload is actually realized by clustering the corresponding semantic segments in the cache, and is conducted simultaneously with the query processing procedure. If a query Q can be answered by a segment S, after Q is cached as a segment, say S_Q, we put S_Q in the same cluster as S belongs to. Though as a result of decomposition and coalescence, the actual contents of the segments might not be the original query results, the semantic relationship history is recorded.

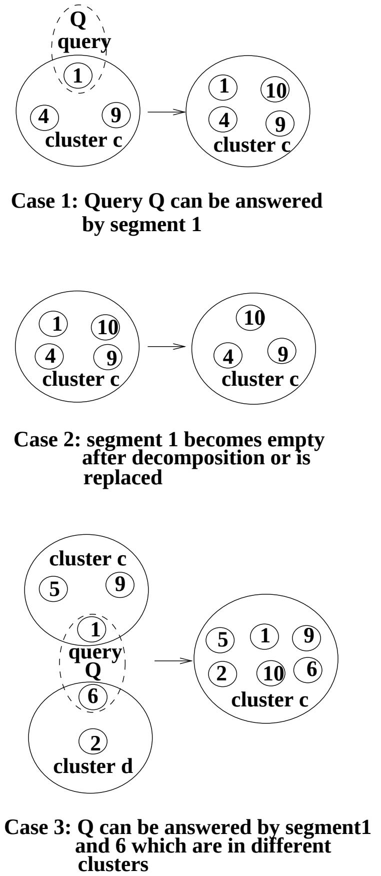

Figure 1: Cluster Maintenance

This construction procedure and other cluster maintenance schemes are illustrated in Figure 1. When a query Q can be answered (perhaps only partially) by segment 1 in cluster c, we cache the answer to Q in cluster c as a new segment, shown as segment 10, and remove from segment 1 the portion which was used to answer Q (Figure 1 Case 1). When a segment becomes empty after decomposition or is discarded from the cache, we remove it from the related cluster (Segment 1 in Figure 1 Case 2). If a query can be answered by segments that belong to different clusters, the clusters get coalesced into one (shown in Case 3). Here, Q helps to find out the semantic adjacency between segment 1 and 6, which is then extended to the clusters they are in. For any pair of segments in a cluster, the corresponding queries either have common resulting tuples, or they are semantically related or adjacent through one or many other earlier queries.

S	SR	SA	SP	SC	STS	Cluster
S1	Stock	{Sname}	20 < Price < 40	2	T1	1
S2	Company	{Cno, Cname}	Business="computer"	4	T2	2
S3	Stock	{Sname, Price}	Price > 40	3	T3	1
S4	Stock	{Sname, Cname}	Price < 20	8	T4	1

semantic cache index

CID	CTS	CSeg
1	T4	S1 → S3 → S4
2	T2	S2

clustering structure

Figure 2: Maintenance Data Structures

Each cluster C is associated with an Id C_{id}, a set of segments linked as a LRU queue, C_{seg}, and a timestamp C_{ts} which is the most recent one among all the S_{TS} of the segments in C_{seg}. In addition to the index described in last section, we also maintain in the cache a cluster structure containing a set of clusters. An example of the cache maintenance data structures is illustrated in Figure 2.

From the cluster structure, we can derive the semantic relationships among queries. If a segment is the only member in its cluster, we say that the corresponding query is not semantically related to the other queries in the workload, or at least the relationship has not been shown till then. Moreover, the characteristics of a workload can also be learned. If there exist a lot of clusters and each contains only a small number of segments, then the workload doesn't have a good degree of semantic locality. In the extreme case when there is only one segment per cluster, the workload doesn't exhibit any semantic locality at all.

Clustering can be used in many ways to improve the semantic cache performance. If we associate each cluster with a single constraint formula, which is the union of the predicates of the segments in it, we can locate the candidate segments to answer a query more efficiently at a cluster level, rather than checking every segment in sequence. The whole procedure is divided into two steps. First, the clusters which might contain the query result are determined. This is done by reasoning among the query predicate and the cluster constraint formula. Then we process the query via these candidate clusters.

Cache admission policy can be built on clustering too. Initially, when the semantic cache is empty, the results of the first queries will always be cached. But for later coming queries, we must decide whether to cache them or not. When a query is requested which is totally unrelated to any clusters in the cache, if space is limited we may not wish to admit it. This is based on the assumption that if a query is not semantically related with previous queries, it may not be related with future queries either. However, strategies must be carefully designed to be able to detect the user attention shifts.

3.3 Two-Level LRU

Traditional cache replacement strategies such as LRU and MRU are based on temporal locality, which is the property that the recently visited items are more likely to be referenced again. In order to keep the data that are most likely to be visited later in the semantic cache, we build the replacement strategy on the semantic locality and let it be able to dynamically adapt to the query access pattern. The approach used is to predict the future access probabilities for the cached segments by analyzing the semantic locality from the query request history. Clustering, which indirectly records the semantic relationship history, provides us an effective way to do that.

The replacement strategy proposed in this paper is called Two-Level LRU, which makes use of both semantic and temporal locality. The first level LRU is carried out at a cluster level, the cluster with the oldest timestamp is selected as a candidate cluster that needs to be further examined. Then, for all the segments in this cluster, we perform the second level LRU and replace those which are least recently accessed until the cache space is sufficient to hold the new query. The Two-Level LRU is described in detail by the following procedure. This assumes that the timestamp of a cluster/segment is updated when it is determined to be needed by a query and before this algorithm is called.

Algorithm: Cache Replacement - Two-Level LRU

while (there are clusters in the cache) {

  clu $\leftarrow$ the cluster with the oldest timestamp in the cache;

  while (there are semantic segments in *clu*) {

    $S \leftarrow$ the semantic segment in *clu* with the oldest timestamp;

    discard S from the cache;

    delete S from *clu*;

    add free space;

    if (free space is enough) return(*Success*); };

  delete *clu* from the cache; };

return(*Fail*)

Based on semantic locality, we know that if one part of a cluster has been recently visited, the other parts are also likely to be accessed soon. Thus we treat a cluster as a whole when choosing segments for replacement. Temporal locality helps to trace the change of user interest. If a cluster hasn't been visited for a long time, it is "cold" and may not be referenced again for a while. This represents a change in the semantic interests of the user. First level LRU aims at locating the "cold" region. It ensures the replacement strategy is able to adapt to the query access pattern. Then the second level LRU selects replacement victims from the chosen cluster, which exploits temporal locality to remove the "coldest" parts from the "cold" region. In Two-Level LRU, semantic locality is assigned a higher priority than temporal locality. For example, if one of the segments in a cluster has recently been visited, the whole cluster is treated to be just visited, hence the rest of segments in this cluster won't be chosen as the replacement victim even though they may have older timestamps than the segments in other clusters. Two-Level LRU still works fine when the query workload doesn't exhibit semantic locality, i.e. there is possibly only one segment in every cluster. In this case, it simply reduces to traditional LRU.

4 Performance Study

We examine the performance of the proposed scheme in a mobile computing environment through a simulation study. Our simulator is implemented in C++ using Csim([CSIM18]). It simulates a simple but typical mobile computing model. In this section, we first present the simulation environment, which includes the simulation model, system parameter settings and workload design. Then we describe the conducted experiments and further analyze the results. The results reported here are preliminary as the simulation studies are still underway.

4.1 Simulation Environment

In this study, the mobile system is modeled as a single server, a single mobile client, and a wireless link between them. We assume the server maintains a complete copy of the database and acts as a database server. All the queries are submitted by the mobile client. The semantic cache is located on the client side. When a query is requested, it is first processed locally and sent to the server for processing only when it can't be answered via the cache. The mobile client and server communicate through a wireless link, with message flows of query requests and query results.

Since the objective of this study is to examine the performance of semantic caching, the server node is not simulated in detail, instead, it is abstracted as a black box. The server takes in query requests and returns query results. The work done at its side is represented by the time elapsed, which is a function of the query result size. In the following, we focus on describing the simulation model of the mobile clients.

4.1.1 Mobile Client Model

The mobile client is simulated to be composed of the following modules: *query generator*, which generates queries according to the workload; *semantic cache query processor*, which processes queries via the semantic cache; *semantic cache manager*, which manages the semantic cache; and *network manager*, which manages the network between the client and the server. We model memory caching on the mobile client side, with a size which will be specified later. In order to maintain fairness to different management strategies, the space overhead for cache management, such as index and the clustering structure, is also counted. Since the management information for semantic caching is maintained at a granularity of semantic segments, which keeps on changing as time passing by, the space overhead is variable. Our approach simply makes sure that the size of management overhead and actual data doesn't exceed the cache size.

4.1.2 System Parameters

Table 1 shows the main parameters used in the simulation study. The first part gives the parameters that specify the physical resources of the modeled mobile system. As we have indicated, server is not simulated in detail, hence the server side parameters are omitted. The client side CPU adopts a FCFS scheduling policy, its speed is described by *ClientMips*. *ClientCache* defines the size of the memory cache at the client side. The wireless network bandwidth is given by *BandWidth*, which has a typical value of 19.2kbps. Database relations, cached semantic segments and cache maintenance data are all physically stored in pages, whose size is specified by *PageSize*. The second part of the System Parameter Table provides the network protocol cost model and the cost of executing operators given in number of instructions, which are taken from [BTJ96]. The wireless network is modeled as a FIFO queue. The

Param	Value	Description
ClientMips	50	mobile client CPU speed (Mips)
ClientCache	128	client side semantic cache size (kb)
BandWidth	19.2	wireless network bandwidth(kb/s)
PageSize	4096	size of data page(bytes)
FixMsgIns	20000	fixed part protocol cost to send/receive a message
PerByteMsg	3	size-dependent part protocol cost to send/receive message
Compare	2	instructions to compare
CopyWord	1	instructions to copy a word

Table 1: System Parameter Settings

cost of sending/receiving a message involves the time-on-wire(transfer time), and the time for protocol including a fixed part(FixMsgIns) and a size-dependent part(PerByteMsg). Only the cost of basic operators is listed in Table 1, for more complicated operations such as predicate computation and reasoning, the cost can be easily derived. The disk paremeters are not described, because in this study, we don't implement a disk cache on the client side, and the server is not simulated in detail.

4.1.3 Workload Design

We use a modified Wisconsin Benchmark([Gray93]) to examine the performance. The benchmark database contains one single relation, R, with 10,000 tuples, each tuple has a size of 200 bytes. For ease of implementation, R consists of 50 4-byte integers instead of the original benchmark scheme with 13 4-byte integers and 3 52-byte character strings. Queries used in this study involve only select and project operations. Size of queries can be changed in the experiments and are specified by *SelRag* and *AttProj*. A fixed portion of queries(*Skew*) has the semantic centerpoint within a *hot region*, the remainder queries are uniformly distributed over the cold area. The size of the hot region is defined by *TupHot* and *AttHot*. Table 2 also gives other parameters to control the experiments. *ClustOn* indicates whether the clustering is turned on. *DisRate* specifies the disconnection ratio, which gives the degree of how often the wireless network is disconnected when a query is requested. Here, we don't simulate the durations of disconnections, instead we always assume that the network is re-connected before the next query.

Param	Value	Description
RelSize	10,000	relation size (tuples)
TupSize	200	tuple size (bytes)
AttSize	4	attribute size (bytes)
TupHot	0.2	percent hot tuples
AttHot	0.2	percent hot attributes
SelRag	100-1000	selectivity (tuples)
AttProj	10	projectivity (attributes)
Skew	0.9-0.5	percent of queries in hot region
ClustOn	T/F	turn on the clustering
DisRate	0,0.2	disconnection rate

Table 2: Workload Parameter Settings

4.2 Experiments

The experiments in this study are designed for two objectives. First, we examine the performance of clustering and Two-Level LRU in normal mobile systems, i.e., in systems when clients and servers are connected. Then we study the effectiveness of our strategies under the situation of disconnection. A series of experiments have been conducted, however due to space limitation, only some representative sets of experiments are presented.

4.2.1 Experiment 1 - When Connected

In this experiment, the wireless link is set to be always connected, i.e. DisRate=0. We compare the performance of Two-Level LRU with the traditional LRU, the primary metric used is response time, while the number of messages transferred on the wireless link and cache management cost are also examined. The presented results were obtained by averaging the results of three runs of simulation. Each run consists of 50 warm-up queries followed by 500 query executions when the actual measurements were taken.

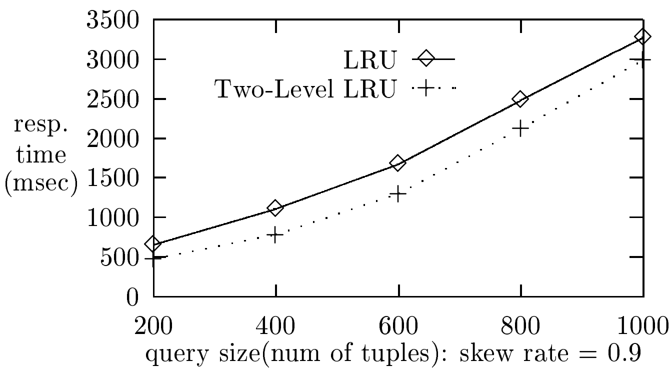

Figure 3: Response Time Result When Skew = 0.9

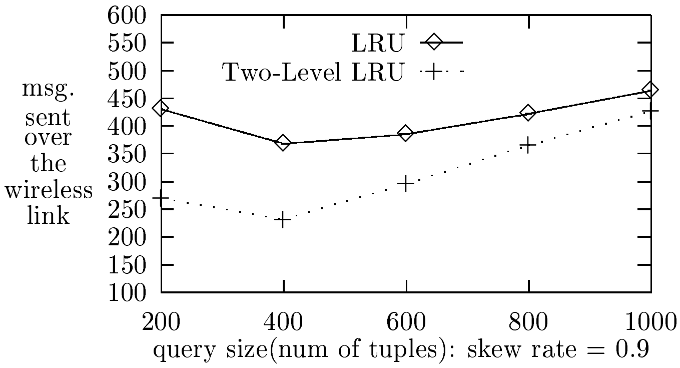

Figure 4: Network Traffic Result When Skew = 0.9

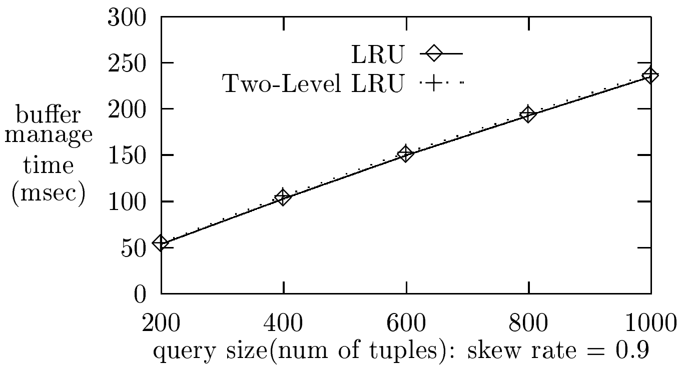

Figure 5: Buffer Cost Result When Skew = 0.9

Figure 3, Figure 4 and Figure 5 show the results obtained when query skew rate is set to 0.9. As shown in the figures, Two-Level LRU completely outperforms LRU. From the query workload description, we know that part of queries always visit the fixed hot region, when this skew rate is high, say 0.9, the workload exhibits a very good semantic locality. Clustering helps to find this locality, a cold region query will soon lose its privilege in replacement when a new hot region query comes, even though it has a very recent timestamp if LRU is used. Built on clustering, Two-Level LRU thus is more effective than LRU in keeping useful data in the cache. Because more data can be obtained locally, the interactions between the client and server are reduced, which can be seen from the metric of the total number of message transferred over the wireless link (Figure 4). This is very important for mobile computing where the network resource is scarce. Moreover, from Figure 5, we observe that the cache maintenance cost for clustering is only slightly higher than that of non-clustering, this is due to two reasons. First, the memory space overhead of clustering is small, second it is easy and cheap to maintain.

The next two graphs (Figure 6 and Figure 7) give

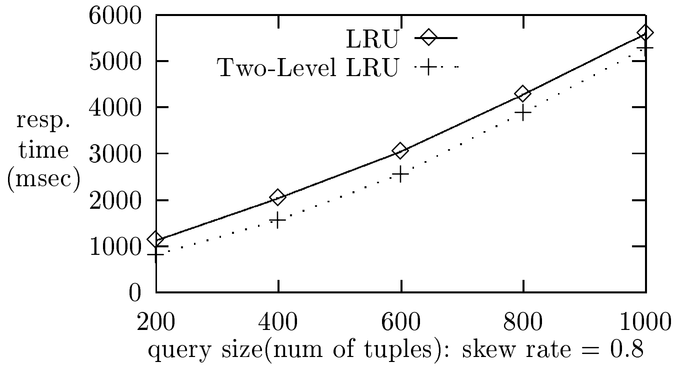

Figure 6: Response Time Result When Skew = 0.8

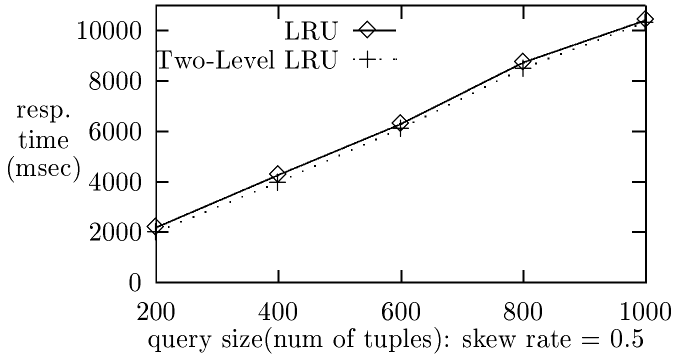

Figure 7: Response Time Result When Skew = 0.5

the response time when skew rates are set to 0.8 and 0.5 respectively. The less the skew rate, the less the degree of semantic locality, and hence the less difference between Two-Level LRU and LRU. When skew rate is 0.5, the performance lines of the two strategies are very close, which strengthens our former assertion: Two-Level LRU reduces to LRU when the workload doesn't exhibit a good degree of semantic locality.

4.2.2 Experiment 2 - When Disconnected

Now we examine the performance under disconnection. The main metric used in this experiment is the percentage of a query result that can be found in the cache during disconnection, which aims at indicating the degree of data availability when a mobile client becomes disconnected. We handle disconnection in this way: when a query is requested when the network is disconnected, we calculate the ratio of how much the portion of the result can be found in the local cache and record that value. Figure 8 shows the performance result when skew and disconnection ratio are set to 0.8 and 0.2 respectively. From the figure, we can see that Two-Level LRU always has a higher degree of data availability than

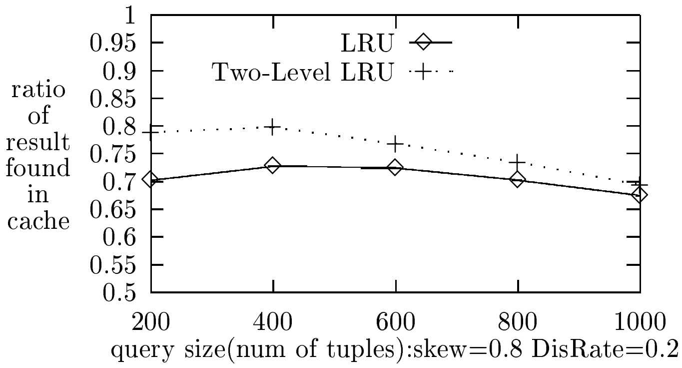

Figure 8: Performance Result from Experiment 2

LRU, this is because Two-Level LRU is more effective in keeping hot data, as we have discussed before. Notice that as the query size increases, the size of cache relatively decreases, thus we have worse performance. The top points of both strategies are observed when queries have a tuple size 400. This is because when queries are too small and the hot region size is unchanged, the subsequent queries become less related, hence the degree of cached data reusability reduces. The same scene can be see in Figure 4 when the network traffic is compared.

5 Conclusions and Future Work

We discover that the performance of a semantic cache depends on the nature of query workloads and further on whether the cache management schemes can adapt to different workloads. The clustering approach has been proposed to study the characteristics of workloads by examining the semantic locality among the queries. With the advantages of low cost and easy of maintenance, clustering can be used as the base for developing effective management strategies. In addition, we have presented the cache replacement strategy - Two-Level LRU, which aims at keeping the most profitable data in the cache with the help of clustering. Because of intrinsic properties, semantic caching is regarded as an ideal cache scheme for mobile computing. The behavior of our strategies in mobile environment has also been illustrated through a set of representative experiments.

For the future research, we intend to work along the following dimensions. First, we plan to apply clustering to query processing as well as other cache management aspects, such as cache admission, candidate segment choosing, etc. Second, we would like to design more kinds of query workloads, analyze them using clustering and further study the performance of our management strategies under them. Moreover, the semantic cache coherency control, which is an very important issue in cache management, will also be investigated in the future study.

References

[BTJ96] Bjorn Thor Jonsson, University of Maryland, *SIUMEI: Design Overview and Class Interfaces*, http://www.cs.umd.edu/projects/dimsum/

[CR94] Chungmin M. Chen and Nicholas Roussopoulos, *The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching*, Proceedings of the EDBT, 1994, pp. 323-336

[CSIM18] Mesquite Software Inc., *CSIM18 Simulation Engine (C++ Version)*

[CSL98] Boris Y. Chan, Antonio Si and Hong V. Leong, *Cache Management for Mobile Databases: Design and Evaluation*, Proceeding of ICDE, February 1998, pp. 54-63

[CTO97] Jun Cai, Kian-Lee Tan, Beng Chin Ooi, *On Incremental Cache Coherency Schemes in Mobile Computing Environments*, Proceedings of ICDE, 1997, pp. 114-123

[DFJ96] Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivatava and Michael Tan, *Semantic Data Caching and Replacement*, Proceedings of the VLDB Conference, 1996, pp. 330-341

[DRSN98] P.M.Deshpande, K.Ramasamy, A.Shukla and J.F.Naughton, *Caching Multidimensional Queries Using Chunks*, Proceedings of SIGMOD, June 1998, pp. 259-270

[GG97] P. Godfrey and J. Gryz, *Semantic Query Caching in Heterogeneous Databases*, In Proceedings KRDB at VLDB'97, August 1997

[Gray93] Jim Gray, *The Benchmark Handbook*, Morgan Kaufmann Publishers Inc., 1993, pp. 272-280

[IK96] Tomasz Imielinski and Henry F. Korth, *Introduction to Mobile Computing*, Mobile Computing, pp. 1-43, Kluwer Academic Publishers, 1996

[KB96] Arthur M. Keller and Julie Basu, *A predicate-based caching scheme for client-server database architectures*, The VLDB Journal, Vol. 5, No. 2, Apr. 1996, pp. 35-47

[RD98] Qun Ren and Margaret H. Dunham, *Semantic Caching and Query Processing*, Technical Report 98-CSE-4, Southern Methodist University, May 1998, http://www.seas.smu.edu/~qren.

[Rou91] Nicholas Roussopoulos, *An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis*, ACM Transactions on Database Systems, Vol. 16, No. 3, 1991, pp. 535-563

[SSV96] Peter Scheuermann, Junho Shim and Radek Vingralek, *WATCHMAN: A Data Warehouse Intelligent Cache Manager*, Proceedings of the VLDB Conference, 1996, pp. 51-62