Using Semantic Caching to Manage Location Dependent Data in Mobile Computing*

Qun Ren
Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275
qren@seas.smu.edu

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

ABSTRACT

Location-dependent applications are becoming very popular in mobile environments. To improve system performance and facilitate disconnection, caching is crucial to such applications. In this paper, a semantic caching scheme is used to access location dependent data in mobile computing. We first develop a mobility model to represent the moving behaviors of mobile users and formally define location dependent queries. We then investigate query processing and cache management strategies. The performance of the semantic caching scheme and its replacement strategy FAR is evaluated through a simulation study. Our results show that semantic caching is more flexible and effective for use in LDD applications than page caching, whose performance is quite sensitive to the database physical organization. We also notice that the semantic cache replacement strategy FAR, which utilizes the semantic locality in terms of locations, performs robustly under different kinds of workloads.

1. INTRODUCTION

Location dependent data (LDD) is the data whose value is determined by the location to which it is related. Examples include local yellow pages, traffic reports, weather information, maps and so on. *A location dependent query* is a query that is processed on location dependent data, and whose result depends on the location criteria explicitly or implicitly specified. Moreover, the result may change as the user changes his location. Although such applications can also be requested in and supported by conventional systems, mobility increases the intensity of the needs, opportunities and challenges. Nowadays, location-dependent applications are becoming more and more popular in the mobile computing environment.

*This material is based upon work supported by the National Science Foundation under Grant No. IIS-9979458

Example 1 - A Continuous LDD Query: *Suppose Jane is traveling through a new city in her car and she wants to find the nearest restaurant. A typical query requested could be "Show me the restaurants within a radius of 5 miles". However, assume that no satisfactory restaurant is found since Jane prefers chinese food, the same query may be issued continuously.*

To improve system performance and deal with disconnection, caching is crucial to LDD applications. In Example 1, Jane gets different restaurant information as she moves to different locations. Furthermore, since Jane asks the same question continuously, there may be a large degree of overlap in the results of consecutive queries. Clearly, continuously evaluating a query from scratch would be very inefficient. If we keep a cache on the mobile unit, part of the new query result could be obtained locally. By doing this, the wireless network traffic is reduced, and the system performance is improved as well.

However, how to choose an effective caching scheme requires further study. Obviously, location plays a distinguished role in location dependent queries, it actually provides an additional, if not the only, semantic criteria to access the data. Hence an LDD query workload is more likely to exhibit a *semantic locality* in terms of the locations, rather than a static *spatial locality*[1] defined by the fixed database physical organization. That is to say, an LDD data item with a location closer to the current position of the mobile user and in the direction of his movement is more likely to be visited in the near future. This unique characteristic makes the traditional page or tuple caching inappropriate to the LDD applications in mobile computing. In addition, the semantic locality among LDD queries is usually highly related with the moving path of the mobile user, e.g. the continuous query issued in Example 1. To dynamically adapt to the query access pattern, the cached contents are required to move as the mobile unit moves. This observation motivates the development of cache replacement strategies built around location and movement.

The idea of *semantic caching* is that the mobile client maintains both the semantic descriptions and associated answers of previous queries in the cache. If a new query is totally an-

[1]Here, spatial locality means that the data items which are physically stored closely are likely to be accessed together.

swerable from the cache, no communication with the server is necessary; if it can only be partially answered, the original query is trimmed and the trimmed part is sent to the server to be processed. Semantic caching is by nature an ideal cache scheme for location dependent applications due to the following reasons. First of all, semantic caching is built on the semantic locality among queries, which just fits the LDD applications where much semantic rather than temporal or spatial locality is exhibited. Secondly, continuous LDD queries can be incrementally processed by semantic caching. With each successive request, a much smaller trimmed LDD query is processed at the server side and only the differences are transmitted over the wireless link. Thirdly, semantic caching makes cache management more flexible. The cache can be managed based on temporal or location information, it could even be managed according to the instructions explicitly given by the mobile user. It is the semantic granularity and the semantic descriptions kept that makes all of these possible. At last, semantic caching also facilitates disconnections. Even though the data at current location can not be obtained, the mobile user might still be able to learn the information for other neighbor locations from the local cache. A page or tuple cache can not offer such functionality, since the cached data there is not associated with any semantic meanings.

This paper is composed of two main parts. The first part of the work includes a formal model to represent moving objects and the definition of LDD queries. From that starting point, we then investigate ways in which a semantic cache can be used efficiently in LDD applications. In Section 2, we first review related research in location dependent data, then examine previous semantic caching and cache management work. We model mobility and define location dependent queries in Section 3. Section 4 gives the strategies of applying semantic caching to location dependent applications. The performance of the proposed scheme is examined through a simulation study in Section 5. Finally, we summarize our work and discuss future research in Section 6.

2. RELATED WORK

One area of research that is related to this paper is about location dependent application modeling, including how to represent moving objects and define location dependent query. Another body of relevant work is the semantic caching scheme and the cache management issues. In this section, we review previous works from these two aspects.

2.1 Modeling Mobility

A formal data model (MOST) to represent moving objects in a database system was introduced in [20]. They treat the position of a moving object as a dynamic database attribute, and express it with three sub-attributes, value, updatetime and function, where the function indicates how the value changes over time. [20] also proposes three types of queries arising in MOST: instantaneous, continuous and persistent. These queries exhibit both temporal and spatial features, since they query moving objects whose values change as time passes by. Moreover, the parties who issue these queries may also move. The problem of how to index moving objects in a database is investigated in [14] and [20]. A straightforward use of spatial indexing is inefficient and infeasible since the spatial index has to be continuously updated when the objects are continuously moving. The previous works discussed so far mainly deal with how to model, store, index and query moving objects in databases. In this paper, we are interested in how to query location dependent data from a moving party. The data are not assumed to be dynamic database attributes.

In [7], location dependent queries are shown to be important and popular applications related to mobility of a mobile environment. They also indicate the research challenges brought about by this new type of application, such as changes to query language and database design. More LDD examples and further discussions about LDD management can be found in [8]. A formal model, which treats location dependent data as database *spatial replicas* tightly coupled with specific *data regions*, is proposed. Then based on this model, research issues such as query processing, caching and transaction management are explored. [8] serves as an excellent review about LDD applications, however, it doesn't provide detailed algorithms and performance study.

[2] implements a location dependent application system, which provides a way for mobile users to access location dependent information from the Internet. Like [8], they also partition LDD data among servers, and propose a network-layer primitive, *nearcast*, for a mobile client to locate and access the specific server covering the data it needs. [2] points out that the cached LDD data can become obsolete due to an explicit update from the server, and because of a move as well. To solve this new location-dependent invalidation problem, for LDD data, they send the associated *scopes* to the mobile client along with the contents. The scope information is sufficient for the client to detect location-dependent validity, just as an attached expiration period is used in checking traditional time-dependent validity.

Our work differs from these previous research in that we focus on location dependent data caching issues. We apply the semantic caching scheme to LDD applications, with every cached item having an attached semantic description. We also examine cache management strategies in the LDD domain.

2.2 Semantic Caching and Cache Management

In what follows, we first briefly review previous semantic caching work from the aspects of definition, organization and operation. Then we look at different strategies in cache replacement.

Semantic caching has been widely used in centralized systems ([4], [18]), client-server environment ([5], [13]), OLAP systems ([6]), mobile computing ([15], [17]) and heterogeneous systems ([9]). All these works cache query results rather than database tuples or pages. The cache is composed of a set of items attached with the related semantic descriptions, which are called *semantic regions* in [5], *semantic segments* in [17] and so on. While logically the cache is always organized using an index which maintains the semantic as well as physical storage information for every cached item, there are various ways to physically store the data. [5] stores semantic regions in tuples, and [17] stores semantic segments in pages. The query processing strategies used in these works are very similar. When a new query comes, it gets split into two disjoint pieces: one that can be locally answered by the cache and one which can not. Only the second part is sent to the database server to be processed. Another important issue is how to partition and coalesce the disjoint parts after query processing. Different coalescing and decomposing strategies are developed in [5], [15] and [16]. The objective of this paper is to apply semantic caching to location dependent applications. We base our work on the cache model proposed in ([16], [17]), which is extended to incorporate the unique characteristics of LDD data.

The commonly used cache replacement strategies are built on temporal locality, such as LRU, MRU and CLOCK. Profit-based or benefit-based replacement schemes are proposed in [6], [13] and [19]. [19] implements an intelligent cache manager in data warehouse systems, where the retrieved sets by queries are cached. They choose the replacement victim by considering its average reference rate, size and the execution cost of the associated queries. Similarly, [13] develops a benefit-based replacement strategy for a semantic cache. [6] 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. Another different replacement policy is used in [3]. The victim is selected according to the access probabilities which are predicted by observing the data access history. Moreover, [5] utilizes *semantic locality* in semantic caching replacement. For each cached *semantic region*, a 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 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. This paper also utilizes semantic information to do replacement. However, since we work in LDD application domain, the semantic locality used is highly related with the moving behavior of mobile users.

3. MODELING LDD QUERY

Location dependent queries usually originate from moving objects, whose locations determine the results of these queries. To effectively manage LDD queries, a mobility model is required to represent the locations and moving behaviors for mobile users. We model the moving behavior of mobile clients from the aspects of locations, speeds and directions. While in general an object can move anywhere in the 3-dimensional space with varied speed and direction, this research assumes that a mobile unit always moves in the 2-dimensional space at a speed that keeps constant during any observed time period and may change from time to time. This assumption makes sense in many cases, e.g. cars moving on highways, planes flying in the sky and so on. To model a moving object, we first define the related basic concepts.

Definition 1 *A* **Location**, *L, is a tuple* (L_x, L_y), *where* L_x *and* L_y *are integers.*

A location denotes an exact position in space. In real life, a location can always be represented in terms of a pair of the absolute latitude and longitude coordinates, which are both float numbers. Without loss of generality, we truncate them into integers in this research. Alternatively, given a reference point in a predefined geographic region, Definition 1 can also be used to stand for relative locations. We assume that a mobile client is equipped with a certain mechanism to obtain its current location. For example, the client may determine its location via GPS or through communicating with the corresponding server. Here, all the details about location tracking are ignored.

Definition 2 *A* **Velocity**, *V, is a vector* $< V_x, V_y >$, *where* V_x *and* V_y *are integers.*

For ease of modeling, we build up a reference coordinate system, where x axis stands for the east and west dimension with the east direction being positive, and y axis represents the north and south dimension with the north direction being positive. To model both the speed and direction of an object moving in a 2-dimensional space, we use a pair of values V_x and V_y, which are the projections of its speed vector on x axis and y axis respectively. Notice that the signs of V_x and V_y, and the ratio of the absolute value of V_x and that of V_y determine the direction of the movement. Some velocity examples are given in Figure 1. Suppose the mobile unit starts from P_1 with the location (1, 1), when it moves at a speed of $\sqrt{2}$ in the northeast direction, its velocity can be expressed as $V_1 =< 1, 1 >$. After two seconds, it arrives at P_2 with the location (3, 3). Then it moves to the west at a speed of 2, and its velocity is $V_2 =< -2, 0 >$. Later on, the mobile unit moves to the south at the velocity $V_3 =< 0, -5 >$, changes its velocity to $V_4 =< 6, -3 >$ at P_4, moves to P_5 and finally arrives at P_6 with a velocity of $V_5 =< 3, 1 >$.

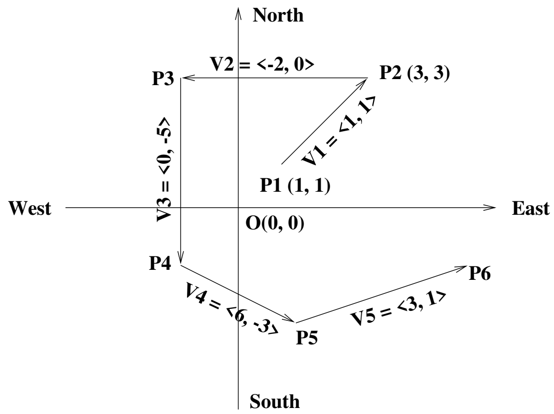

Figure 1: The Velocity Examples

Velocity helps to predict future locations of moving objects. Let L = (L_x, L_y) stand for the location of a mobile client M at time T, also suppose that M moves at a velocity V = $< V_x, V_y >$ and keeps this velocity during the period $[T, T + \delta T]$, then M's location at time T+δt, where $\delta t \leq \delta T$, is expressed as $L_{\delta t} = (L_x + V_x \times \delta t, L_y + V_y \times \delta t)$. This can be illustrated by a scenario in Figure 1 when the object moves from P_1 to P_2.

At a certain point in time, the status of a moving object can be specified by its location and the velocity of the movement.

As time passes by, the object may change its moving behavior. Hence, the status of a mobile unit is modeled to be always associated with a particular timestamp.

Definition 3 *The* **Status**, *of a mobile unit, M, is a tuple* (M_T, M_L, M_V), *where* M_T *is a timestamp,* M_L *is M's location at* M_T *and* M_V *is M's velocity at* M_T.

For example, in Figure 1, assume that the mobile unit starts at time T_1 at location $P_1(1, 1)$, then arrives at $P_2(3, 3)$ at time T_2 and changes its velocity there. According to Definition 3, its status at P_1, P_2 can be expressed as (T_1, $P_1(1, 1)$, $V_1 < 1, 1 >$), (T_2, $P_2(3, 3)$, $V_2 < -2, 0 >$) respectively. Keeping track of the status of a mobile user helps to examine his location dependent query workload, thus facilitates cache management. We can store in the cache those data that are very likely to be visited in the future by making the replacement strategy dynamically adapt to the query access pattern. Also, useful information can be prefetched based on the user profile in which the future status are specified.

The key issue in modeling location dependent queries is how to specify the location related select predicates. Our strategy is to treat the location criteria as normal conditions. Before processing LDD queries, they must be bound to a precise location. This process adds the location attribute values to them.

Example 2 - Modeling an LDD Query: *Suppose there is a hotel relation with attributes* **name, price, vacancy, xposition, yposition**, *and the mobile unit M issues a query Q = "Give me the names of the hotels within 20 miles whose prices are below \$100." at location L. Then the predicate of Q,* Q_P *can be represented as* Q_P = (price < 100) $\wedge$ (L_x - 20 $\leq$ xposition $\leq$ L_x + 20) $\wedge$ (L_y - 20 $\leq$ yposition $\leq$ L_y + 20). *It is evident that* Q_P *is not a direct select condition, and Q can be answered only when the value of L is given. Assume that currently L is (10, 30), then* Q_P *is converted into* $Q_P{}'$ = (price < 100) $\wedge$ (-10 $\leq$ xposition $\leq$ 30) $\wedge$ (10 $\leq$ yposition $\leq$ 50). $Q_P{}'$ *could have a different expression as M moves to another location. Sometimes, even if M stays at the same location, it may ask for hotel information at another place. In this case,* $Q_P{}'$ *also changes because L changes.*

We consider select and project LDD queries in this research, and assume that each query predicate involves only one location variable. However, the proposed model can be easily extended to handle other more complicated queries, which is an important direction of future work. LDD queries are constructed with two kinds of predicates: traditional location unrelated predicates and location related ones. Before formally defining an LDD query, we first discuss these two kinds of predicates and other related concepts.

Suppose that the database being examined, D, consists of a number of base relations R_1, R_2, ..., R_n, namely, D = $\{R_i$, 1 $\leq$ i $\leq$ n$\}$. Further let A_{R_i} stand for the attribute set of R_i, and A represents the attribute set of the whole database, i.e. A = $\cup A_{R_i}$. Then, we have the following definitions.

Definition 4 *Given a database* $D = \{R_i\}$ *and its attribute set* $A = \cup A_{R_i}$, *a* **Compare Predicate** *P is of the form P = a op c, where* $a \in A$, $op \in \{\leq, <, \geq, >, =\}$, *c is a constant in a specific domain.*

Definition 5 *A* **Location Variable**, *LV, is a tuple* (LV_x, LV_y), *where* LV_x, LV_y *are integer variables.*

Definition 6 *Given a database D=*$\{R_i\}$, *its attribute set A=*$\cup A_{R_i}$ *and a location variable LV, a* **Location Compare Predicate** *with respect to LV, LCP, is of the form LCP = a op* (LV_x+c) *or LCP = a op* (LV_y+c), *where* $a \in A$, $op \in \{\leq,<,\geq,>,=\}$, *c is a numeric constant. A* **Location Predicate** *with respect to LV, LP, is a disjunction of conjunctions of compare predicates and location compare predicates with respect to LV.*

Look at Example 2 again. According to the above definitions, we know that the predicate (price < 100) in Q_P is a compare predicate which is not related with a location, and (xposition $\leq$ L_x + 20) is a location compare predicate with respect to L. Moreover, Q_P itself is a location predicate. Since an LDD query can be processed only when a certain location is given, we now examine the concept of location binding.

Definition 7 *Given a location L, a location variable LV, and a location predicate LP with respect to LV, the procedure of assigning* L_x *to* LV_x, L_y *to* LV_y *is called the* **Location Binding** *of LV with L. Furthermore, the procedure of performing location binding for every LV in LP with L is called the* **Predicate Location Binding** *of LP with L.*

For ease of expression, in this paper, we denote the predicate location binding procedure of LP with L as **Loc_Bind**(LP, L). Consider Example 2, $Q_P{}'$ is actually obtained through a predicate location binding on Q_P with L(10, 30), i.e. $Q_P{}'$ = Loc_Bind(Q_P, L). An LDD query is uniquely defined when both its predicate and bound location are specified, thus we have Definition 8.

Definition 8 *Given a database* $D = \{R_i\}$ *and its attribute set* $A = \cup A_{R_i}$, *a* **Location Dependent Query** *Q is a tuple* $(Q_R, Q_A, Q_P, Q_L, Q_C)$, *where* $Q_R \in D$, $Q_A \subseteq A_{Q_R}$, Q_P *is a location predicate,* Q_L *is a location,* $Q_P{}'$ = *Loc_Bind(*Q_P, Q_L*), and* $Q_C = \pi_{Q_A}\sigma_{Q_P{}'}(Q_R)$.

4. SEMANTIC CACHING IN LDD

A semantic caching model which stores LDD data is first presented in this section, then we investigate how to process LDD queries through such a cache. At last, we examine how to effectively manage the semantic cache by taking into account the unique characteristics of LDD applications.

4.1 LDD Cache Model

From the definition, we know that an LDD query Q is expressed by the tuple (Q_R, Q_A, Q_P, Q_L, Q_C), with the first four elements specifying the associated semantic information and the last one representing the result. Correspondingly, we use the same tuple (S_R, S_A, S_P, S_L, S_C) to describe an *LDD semantic segment* S, which is a cached LDD query. Among these, S_R and S_A define the database relation and attributes involved in computing S, S_P indicates the select condition that the tuples in S satisfy, S_L gives the bound location and S_C represents the actual content of S. The whole cache is then defined to be composed of a set of such segments. This model differs from the basic semantic caching scheme of [16] in two aspects. First, S_P is a location predicate. Second, the bound location S_L is also maintained in the cache.

Physically, we store the LDD semantic segments by paging them. Each segment is stored in one or multiple linked pages, and is associated with a pointer pointing to its first page in the memory/disk cache. Notice that each page now contains LDD query results rather than database tuples. Meanwhile, the cache is logically organized through an index which maintains the semantic descriptions as well as physical storage informations for every cached segment. The index structure is consistent with the definition of LDD query. In addition to the five basic components of a segment, we might add other items for maintenance reasons, such as the last-visited timestamp which is used for cache replacement. The LDD cache index is more clearly illustrated through the following Example 3.

S	S_R	S_A	S_P	S_L	S_C	S_{TS}
S_1	Hotel	Hname	$(L_x - 5 \leq hxpos \leq L_x + 5) \wedge (L_y - 5 \leq hypos \leq L_y + 5)$	10,20	2	T_1
S_2	Rest.	Rname, Type	$(L_x - 10 \leq rxpos \leq L_x + 10) \wedge (L_y - 10 \leq rypos \leq L_y + 10) \wedge (6 \leq sched \leq 9)$	-5,15	5	T_2
S_3	Hotel	Hname, Vacancy	$(L_x - 5 \leq hxpos \leq L_x + 5) \wedge (L_y - 5 \leq hypos \leq L_y + 5) \wedge$ (Price<100)	-5,-20	8	T_3

Table 1: Example of LDD Semantic Cache Index

Example 3 - An LDD Cache Index: *Consider a yellow page database with two relations: Hotel(Hno, Hname, Price, Vacancy, hxpos, hypos) and Restaurant(Rno, Rname, Type, Schedule, rxpos, rypos). Suppose that the mobile unit M asks the following questions on his way, whose results are cached afterwards.*

- *At time T_1 location (10, 20), M asks query Q_1: "Give me all the names of the hotels within 5 miles", and then keeps on driving to the southwest. The result of Q_1 is cached as segment S_1.*
- *After it arrives at location (-5, 15) at time T_2, M asks query Q_2: "Give me the names and types of the restaurants within 10 miles that open from 6:00pm to 9:00pm", and then keeps on driving to the South. The result of Q_2 is cached as segment S_2.*
- *Finally at time T_3 location (-5, -20), M asks query Q_3: "Give me the names and vacancy information for hotels within 5 miles whose prices are below \$100". The result of Q_3 is cached as segment S_3.*

Assume that the first pages of S_1, S_2 and S_3 are 2, 5 and 8 respectively, the index is shown as Table 1.

4.2 LDD Query Processing

As discussed before, the result of an LDD query is not only determined by the selection and projection conditions specified, but also related with the given bound location. Therefore, the first step of LDD query processing is to perform the corresponding predicate location binding. After that, an LDD query is converted into a normal database query and can be processed from the local cache using the strategies similar to those proposed in [16].

Since an LDD cache is composed of a set of LDD semantic segments, we examine the query processing problem from two aspects: how to process a query via a single segment and how to compute it from the entire cache. The relationship between an LDD query Q and an LDD semantic segment S can be determined from the semantic specifications associated with them. However, to reason among the predicates, we must conduct predicate location binding on both Q and S. If S is known to contain a part or the whole result of Q, we divide Q into two parts through a procedure called *query trimming*: a *probe query* which specifies the portion of Q satisfied by S and a *remainder query* which defines the portion of Q not found in S. Sometimes, the probe query can not be processed from S because the attributes needed to further qualify Q's result are not found in S, where only the projected attributes of S are stored. The solution of [16] is to always keep the key attributes in every cached segment and base on them to append the segments with the relevant missing attributes, which are fetched from the server using *amending queries*. As LDD query predicates often involve the location attributes, to reduce the number of the amending queries, a new heuristic is to also maintain the location attributes in LDD semantic segments even when they are not projected. This is a time and space tradeoff: extra cache space is used to reduce the wireless network delay and cost.

While every individual segment may contain only a small part of a query result, they can combine together to generate a much bigger part of or even the whole result. Therefore, after an LDD query is trimmed by the first segment, we let the remainder query be further trimmed by the next candidate segment that also contains the query result. 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 to be processed. At last, the result of the LDD query is obtained by coalescing every partial result. To efficiently locate the candidate segments, heuristics can be proposed based on the cached bound locations. For example, we may examine the segments according to the distances between their bound locations and the query's bound location: the closer the distance, the sooner the segment is examined.

Another interesting issue related with query processing is how to handle the scenario after an LDD query Q is trimmed by an LDD segment S. Three disjoint parts are left after a query trimming: 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, Q's results are always coalesced together and cached as a new LDD segment. Meanwhile, S gets decomposed accordingly to avoid duplicates. The portion of S not contained in Q is further divided into two parts, the *horizontal LDD subsegment* S_h and the *vertical LDD subsegment* S_v. Since a location dependent query workload tends to exhibit an intrinsic semantic locality interms of locations, the bound location of an LDD segment is a special semantic metric which indicates its relationship with other segments and future queries. As part of S, S_h and S_v should keep that indicator. Thus we associate them with the bound location S_L.

4.3 LDD Cache Management

Since a semantic cache organizes data by semantic metrics, it makes cache management more flexible. Different strategies can be developed to adapt to the requirements explicitly or implicitly specified by the mobile user. When user provided information is not available, strategies need to be carefully designed to effectively manage an LDD cache. As we discussed before, in most cases, the LDD query access pattern is tightly associated with the movement of the mobile user. Hence it is reasonable to incorporate the impact of mobility in LDD cache management. Before describing the detailed algorithms, let us first look at an example.

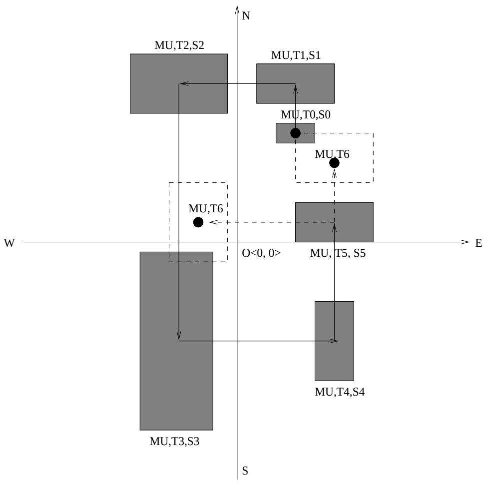

Figure 2: An Example for FAR

Example 4 - Cache Replacement Scenarios: *Figure 2 shows the moving path of a mobile user MU. Suppose the current timestamp is T_5, and the LDD cache has stored the results of queries submitted by MU at time T_0 through T_4, with $T_0 < T_1 < T_2 < T_3 < T_4 < T_5$. T_6 is the timestamp when the next query is expected to come, the predicted MU's location is also shown in Figure 2. Also assume that the query result obtained at T_5 needs to be cached and there is not sufficient space, a victim must be chosen to be replaced. The segment S_0 which stores the result of the query submitted at T_0 would be discarded if an LRU policy is used. However, it shouldn't be replaced since it may soon be visited by the MU at time T_6, just as indicated in Figure 2. This can be derived from the current location of MU and the location associated with S_0. Furthermore, the moving direction is another important factor in replacement. If MU moves to north at T_5, it makes sense to replace S_3. If MU moves to west instead, S_3 can not be discarded.*

An optimal replacement scheme can be found with totally accurate apriori knowledge, since we may simply discard those segments which will not be used during the trip. Otherwise, we can only work on a locally optimal plan using the known information about the user's current status, i.e. his location, moving speed and direction. A mobility-based semantic cache replacement policy was first proposed in [5]. They utilize a semantic value function, the *directional Manhattan distance function*, to calculate a value for each cached semantic region based on its Manhattan distance from the user's current location. Then they discard regions according to the values computed. We extend their work in three aspects and develop an LDD semantic cache replacement policy *FAR (Furthest Away Replacement)*. First, to compute Manhattan distances in [5], the center of each region must be determined. In our case, when a semantic segment is decomposed into two disjoint parts, it is difficult to define its center, let alone calculate it. Hence we use the location information attached to each segment, which is simpler and more efficient. Second, [5] calculates Manhattan distances using estimated weights which are difficult to determine most of time. Instead, we derive the relationships between the cached segments and the current query using the mobile user's current status, which is more accurate and reasonable. Third, [5] does not use the knowledge concerning the moving direction of the mobile user to do replacement. Here we further classify segments into two sets, those segments which are not in the direction of movement will be discarded first.

FAR chooses replacement victims according to the current status of the mobile user. Those segments which are not in the moving direction and are furthest from the user will be discarded first, as we believe that they won't be visited in the near future. Given a mobile user *M*, its status (M_T, M_L, M_V) and an LDD segment *seg*, we tell whether *seg* is in *M*'s moving direction in the following way. According to *M*'s status, we compute a M_{fL}, which is the anticipated *M*'s future location at time $M_T + \delta t$ where δt is a predefined small number. If *seg* is closer to M_{fL} than to M_L, then *seg* is in the moving direction. Otherwise, it is not. Correspondingly, the segments are then divided into two sets: *In-Direction* and *Out-Direction*. The victim is always chosen from the *Out-Direction* set. Once that set becomes empty, the furthest segment in the *In-Direction* set will be replaced. This whole procedure is described in the following algorithm FAR(C, M), where C is an LDD cache and M is a mobile user.

Algorithm FAR(C, M) {

In-Direction $\leftarrow$ NULL;

Out-Direction $\leftarrow$ NULL;

δ t $\leftarrow$ a predefined small number;
$M_{fL} \leftarrow M$'s future location at time $M_T + \delta t$;
for every segment *seg* in C {
  if Distance[2](seg_L, M_{fL}) $\leq$ Distance(seg_L, M_L)
  then *In-Direction* $\leftarrow$ *In-Direction* + {*seg*};
  else *Out-Direction* $\leftarrow$ *Out-Direction* + {*seg*}; }
while (*Out-Direction* != Empty) {
  seg $\leftarrow$ the segment in *Out-Direction* which is the furthest from *M*;
  discard *seg* from *C*;
  remove *seg* from *Out-Direction*;
  add free space;
  if (free space is enough) return (*Success*); }
while (*In-Direction* != Empty) {
  seg $\leftarrow$ the segment in *In-Direction* which is the furthest from *M*;
  discard *seg* from *C*;
  remove *seg* from *In-Direction*;
  add free space;
  if (free space is enough) return (*Success*); }
return (*Fail*); }

5. PERFORMANCE STUDY

We examine the performance of the semantic caching scheme and its replacement policy FAR through a simulation study. Our simulator is implemented in C++ using CSIM ([1]). Prior to presenting the experiment results we discuss the simulation design.

5.1 Workload Design

At present, there are no existing benchmarks for location dependent applications. To evaluate the performance of different caching schemes and replacement strategies, specific LDD query workloads must be properly designed. In this study, we design workloads along three dimensions. The first dimension is the database, which is the basic component and whose physical organization directly impacts the performance. Secondly, since we are dealing with LDD queries, the changes of locations, namely, the moving path of mobile users, is another important design issue. Thirdly, we specify the characteristics of queries from the aspects of size, type and relationships with locations. All the parameters related with workload design are listed in Table 2.

Database Design The database used in this study contains one single relation *R*. Each tuple of R has *NumAtt* attributes, each attribute is 4 bytes long. Two of the attributes x and y simulate the X and Y coordinates in a 2-dimensional space and take values from $[1..MAX_x]$ and $[1..MAX_y]$ respectively. Also we assume that there is always a tuple in R for each pair of (x, y). Hence the database size is determined by the values of MAX_x, MAX_y and NumAtt. To study the impact of database organization on the performance,

[2] Any distance measure could be used here. However, we assume the distance between $L_1(L_{1_x}, L_{1_y})$ and $L_2(L_{2_x}, L_{2_y})$ is defined as Distance(L_1, L_2) = $\sqrt{(L_{1_x} - L_{2_x})^2 + (L_{1_y} - L_{2_y})^2}$.

Parameter	Description	Value
MAX_x	the max value that x can take	400-600
MAX_y	the max value that y can take	400-600
NumAtt	how many attributes per tuple	10
Index	indexing situation on R	non or column-Index on x
Cluster	clustering situation on R	Non or wise Scan
TotalQuery	total queries generated	500
Movement	movement type	1, 2, 3
Speed	absolute move speed of mobile user	16-20
XScope	move how far along x-dimension	100-600
YScope	move how far along y-dimension	80-600
StartLoc	start location	(20, 20)
StartSpeed	absolute start speed	16-20
StartDir	start move direction	N,S,E,W
SelSz	selection range	20
ThinkTm	interval between consecutive queries	1
OverlapRate	overlap ratio between consecutive queries	0-0.2

Table 2: Workload Parameter Settings

two database indexing/clustering cases are simulated. The database is neither indexed nor clustered in the first case, while under the second scenario, it is both indexed on x and clustered through a *Column-wise Scan* function ([11]).

Moving Path Design We assume that the mobile user keeps on asking LDD queries while on the move, hence his moving path actually indicates how the semantic locality among queries looks like. In this study, we develop three different movement types which are commonly observed in our daily life. To simplify the design, only four moving directions, namely, North, South, East and West, are allowed. Also we assume that the mobile user moves at a constant speed. Given a starting status, the next status is generated according to the definition of the specific movement type. In what follows, we give the design details.

- **Workload 1 - OneWay Trip** This is the simplest case. The mobile user moves in a line keeping the direction and speed specified by the starting status, he finishes moving when he hits the boundary of the database or has submitted enough queries.

- **Workload 2 - Round Trip** In this case, the trip is composed of two rounds. The first round is very similar to the OneWay trip, the only difference is that when the mobile user hits the boundary or finishes half of the queries, he changes to the opposite direction and begins the second round. The mobile user's moving behavior in the second round is the same as the OneWay trip.

- **Workload 3 - Random Trip** The random trip workload allows the mobile user to move randomly in a region, the next moving direction is randomly chosen from {North, South, East, West} with an equal probability. In order to control the scope of moving, we specify the boundary of the region using two parameters *XScope* and *YScope*, which indicate how far the user can move along x-dimension and y-dimension respectively. The workload ends when the mobile user has submitted enough queries.

LDD Query Design We consider selection-only LDD queries, where the selection condition is a disjunction of conjunctions of Location Compare Predicates. Each LDD query forms a SelSz $\times$ SelSz rectangle whose center is the mobile user's current location, this defines its selection predicates and specifies its size. We assume that the mobile user requests a new query after a time interval *ThinkTm*, and each pair of consecutive queries overlap with each other at a rate *OverlapRate*. We also assume that the user changes his status after a query is submitted. Hence the absolute moving speed is given by the three parameters SelSz, ThinkTm and OverlapRate.

5.2 Simulation Model

Our simulation model is composed of a single server, a single mobile client, and a wireless link between them. The server maintains a complete copy of the database and acts as a database server, while all the LDD queries are submitted by the mobile client. We also assume that there exists either a page cache or a semantic cache on the mobile client side.

Parameter	Description	Value
ServerMips	server CPU speed(Mips)	50
ClientMips	mobile client CPU speed(Mips)	50
ClientCache	client side cache size(kb)	128
BandWidth	wireless network bandwidth(Mb/s)	1
PageSize	size of data page(bytes)	4096
FixMsgIns	number of CPU instructions to send/receive a message (fixed part protocol cost)	20000
PerByteMsg	number of CPU instructions to send/receive message (size (related part protocol cost)	3
StartIO	number of CPU instructions to start an IO operation	5000
Compare	number of CPU instructions to compare	2
CopyWord	number of CPU instructions to copy a word	1
DiskTm	time to read a page from disk(ms)	12

Table 3: System Parameter Settings

Mobile Client Model To compare the performance of page and semantic caching in LDD applications, two types of client models are simulated: page caching client and semantic caching client. Both clients have the same *Query Generator* module which generates LDD queries according to the workload, and *Network Manager* module that manages the wireless network. However, they differ in query processing and cache management. The other modules of a page caching client include: *Cache Manager*, which manage the memory page cache; *Query Processor*, which processes the query with the data shipped from the server; and *File Manager*, which helps to determine physical locations for required data. If there is an index on the attribute queried, the client first access the index and then access the qualifying data pages. If there is no index, the client has to scan the whole database. For a semantic caching client, the *Semantic Cache Manager* manages the memory semantic cache, and the *Semantic Cache Query Processor* processes LDD queries via the cache. In order to be fair for these two schemes, the space overhead for cache management, such as the Buffer Control Blocks of page caching ([10]) and the index of semantic caching, is also counted.

Server Model The server receives and processes messages from the mobile client. For a page caching client, when the client sends a page request to the server, the server will prepare that page and send it back to the client. In this case, the message flow over the wireless link is page request/page. On the other hand, for a semantic caching client, when the client sends a query to the server, the server processes it locally and sends the result back. The message flow in this case is query request/query result.

System Parameters Table 3 shows the primary parameters used in the simulation study. The first part gives the parameters that specify the physical resources of the modeled mobile system. Both the server and client CPUs adopt a FCFS scheduling policy, and their speeds are described by *ServerMips* and *ClientMips* respectively. *ClientCache* defines the size of the memory cache at the client side. The wireless network bandwidth is given by *BandWidth*. Database relations, cached pages and semantic segments are all physically stored in pages, whose size is specified by *PageSize*. The second part of the table provides the cost model used in the simulation, which is mainly taken from [12]. The wireless network is modeled as a FIFO queue. The 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*). *StartIO* gives the CPU cost for starting a disk I/O operation, and *DiskTm* specifies how long it takes to read a page from the disk. Only the cost of basic operators is listed in Table 3, for more complicated operations such as predicate computation and reasoning, the cost can be easily derived.

5.3 Experiments

The experiments presented are designed for two objectives. First, we study the performance differences between page and semantic caching in LDD applications. Second, we compare our semantic caching replacement strategy FAR with conventional ones. We do not compare FAR to the mobility-based semantic cache replacement policy proposed in [5] as their proposal was preliminary with many implementation details omitted. We run experiments varying the parameters such as database size, database physical organization, movement type, movement range etc. The primary metric used is response time, which is the time from the submission of the query to the time when its result is obtained. In general, this involves three parts: the client processing time, the server processing time and the time spent on the wireless link. All results shown are generated by averaging the results of three runs of simulation.

5.3.1 Page Caching vs. Semantic Caching

This set of experiments studies the effect of database and movement type on the performance of page and semantic caching. The results are shown in Figures 3-5.

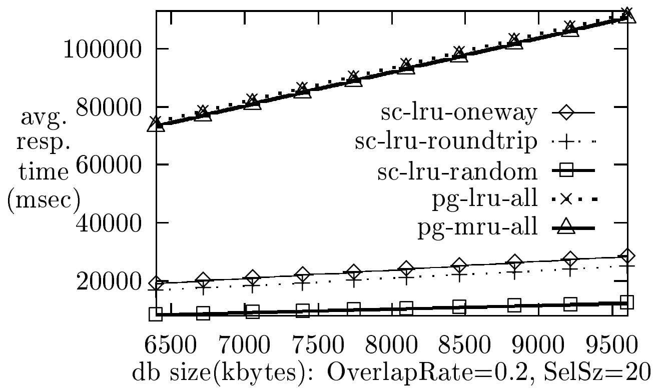

Figure 3: page vs. semantic caching when database is neither indexed nor clustered

Figure 3 gives the response time of the two caching schemes under different workloads when the database is neither indexed nor clustered. Here, OverlapRate is set to 0.2 and SelSz is set to 20. Also we let XScope be 400 and YScope be 80 for the random trip case. It can be seen that the performance of page caching is independent of movement, and becomes worse as the database size increases. This is because when there is no index, the query access path is always a file scan, and more pages have to be scanned as the database becomes bigger. The fixed access path of page caching also makes MRU perform better than LRU. However since the cache size is very small compared to the database size, MRU is only slightly better. The performance of semantic caching varies under different movement types. The random trip in the 400 × 80 range shows the best performance, since it exhibits the highest degree of locality. Semantic caching is less sensitive to the database size, as only part of an LDD query is processed by the server via a file scan. The most important observation from Figure 3 is that semantic caching completely outperforms page caching. This is due to the highly reduced wireless network traffic in a semantic cache case, where only the required data are transferred.

Figure 4 compares the two schemes when the database is indexed on x and is clustered using the Column-wise Scan mapping function. All the other parameters remain unchanged. We can see that the performance of page caching is greatly improved. The page client can first locate the necessary data pages via the index and fetch only those pages from the server, thus the network traffic is substantially reduced. Since now the access path is not a fixed file scan,

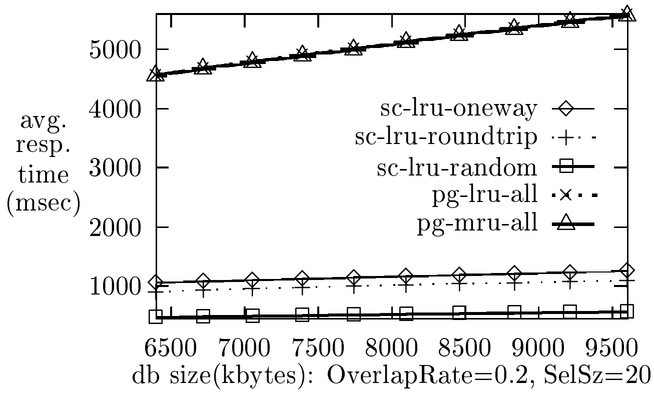

Figure 4: page vs. semantic caching: index on x, column-wise scan clustering

the difference between LRU and MRU is very small. An interesting thing is that even in this case, the movement type still has no impacts on page caching. This is due to two reasons. First, since the database is only indexed on x, there are still a lot of pages to be visited for each query. Second, the workloads used, which are oneway-trip, round-trip along x-dimension and random trip, do not match with the Column-wise Scan database clustering. Hence there are only a small percentage of overlapped pages between consecutive queries, which results in a bad cache hit rate. Many pages still must be fetched over the network. These also explains why semantic caching outperforms page caching in this case too.

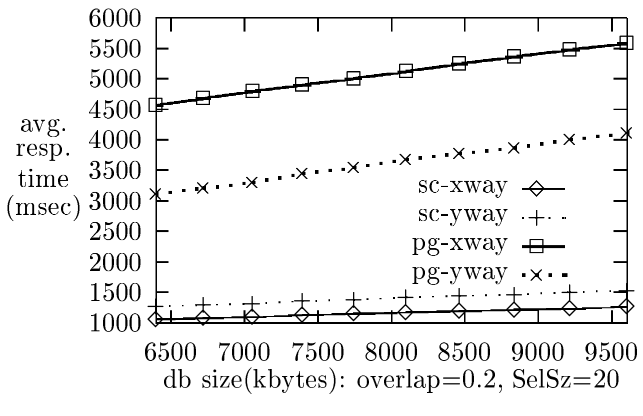

Figure 5: impact of database physical organization on page caching: index on x, column-wise scan clustering

We further study the sensitiveness of page caching to database organization in Figure 5. Workload 1 is used in this case. We let the mobile user move along x-dimension and y-dimension respectively, all the parameter settings are kept the same as before. It can be seen that when the mobile user moves along y-dimension, the performance of page caching is much better than the case when he moves along x-dimension. The reason is very straightforward, when it is a y-dimension moving, the consecutive queries share the same predicate on x, hence they access the same set of data pages when the database is indexed on x and clustered using a Column-wise Scan. As part of the data can be found in the cache, the performance is improved. However, the x-dimension moving couldn't benefit from this organization. Notice that things are totally different for semantic caching: the performance of x-way is better than that of y-way. This is due to two reasons. First, a smaller set of data pages are visited at the server side for x-way queries since their predicates on x get trimmed. Second, we assume that the server doesn't contain any data pages in the memory buffer for a query when it comes.

From this set of experiments, we notice that semantic caching completely outperforms page caching due to the reduced wireless network traffic. While the performance of page caching is quite sensitive to the database physical organization, the effectiveness of semantic caching is related with the nature of workloads. There might exist other more efficient database clustering approach for page caching, but one thing is for sure, the clustering approach only works well when the query workload matches with it. Hence, it is safe to say that semantic caching is more flexible and effective for use in LDD applications.

5.3.2 Semantic Cache Replacement Strategies

In this set of experiments, we study semantic cache replacement strategies. We compare the performance of LRU, MRU and FAR under different workloads.

MoveType	OverlapRt	LRU	MRU	FAR
one-way	0	1865ms	1865ms	1865ms
	0.1	1686ms	1686ms	1686ms
	0.2	1507ms	1507ms	1507ms
round trip	0	1675ms	1675ms	1675ms
	0.1	1520ms	1494ms	1494ms
	0.2	1371ms	1351ms	1351ms

Table 4: Response Time of LRU, MRU and FAR under workload 1 and workload 2, MAXx=MAXy=600

Table 4 illustrates the results for different cases with varied OverlapRate. For the oneway-trip, all the approaches perform the same, the performance simply depends on the overlap rate between consecutive queries. This is reasonable, since this workload doesn't exhibit any kind of locality. There are only small differences between the replacement strategies in the round-trip case. LRU keeps the most recently used data in the first round, which will be visited immediately on the way back. MRU keeps the least recently used data in the first round, which will be guaranteed to be visited in the second round. Moreover, FAR always discards those unuseful data. Hence the three approaches have very similar cache hit rate. Since there is no much space for improvement in these two cases, in the following, we focus on examining the third workload - random trip.

Figure 6 compares the three semantic cache replacement strategies when the random moving path has a good degree

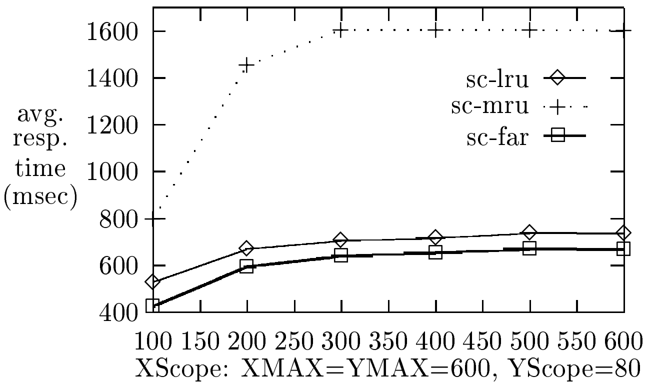

Figure 6: comparing LRU, MRU and FAR under workload 3 - random trip, OverlapRate = 0

of locality. The database parameters MAX_x and MAX_y are both set to 600. The x-axis of Figure 6 stands for the moving range in terms of XScope, while YScope is limited to 80. In order to truly examine the impact of moving path on the performance, OverlapRate is set to 0. Clearly, the performance of MRU is the worst, since it is not effective in utilizing the locality among the moving path. We can also see that FAR outperforms LRU 10% in average, this demonstrates that FAR is more efficient in examing the semantic locality in terms of locations.

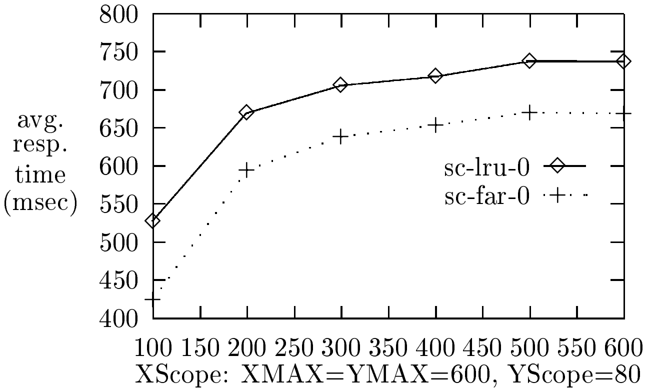

Figure 7: comparing LRU and FAR under workload 3 - random trip, OverlapRate = 0

Since MRU is not comparable to both LRU and FAR, we only study LRU and FAR in the next experiments. Figure 7 and 8 illustrate the results for LRU and FAR with different overlap rates. The objective of these experiments is to find out the factors that impact the performance of LRU and FAR respectively. We keep the same parameter settings as Figure 6. Figure 7 gives the results when OverlapRate is set to 0, while Figure 8 shows the results when OverlapRate is set to 0.1. FAR outperforms LRU in both cases, this is because when YScope is set to 80, the moving path of the mobile user always exhibits a good degree of lo-

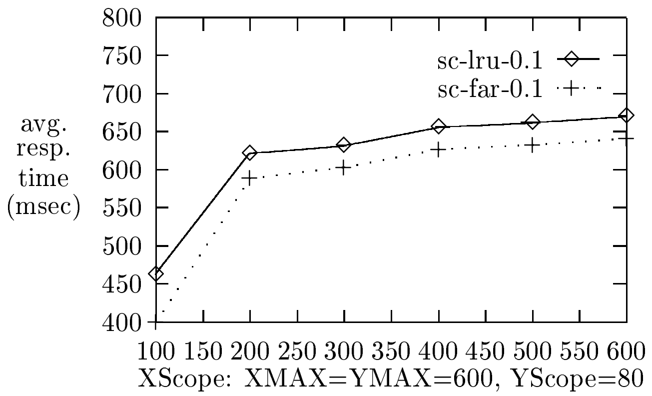

Figure 8: comparing LRU and FAR under workload 3 - random trip, OverlapRate = 0.1

cality. Notice that when the overlap rate is increased from 0 to 0.1, the performance of both strategies are improved, however, LRU shows a relatively better improvement. This demonstrates the fact that while LRU makes use of temporal locality, FAR utilizes the semantic locality in terms of locations. The results shown in Table 5 further strengthens the above assertion.

XScope×YScope	Overlap	LRU	FAR	ratio
100 × 80	0	528	425	19.5%
600 × 600	0	1039	991	4.6%
100 × 80	0.1	463	400	13.6%
600 × 600	0.1	963	948	1.5%

Table 5: Analyze the Performance of LRU and FAR

In some cases, LRU behaves extremely poorly. Table 6 shows such a scenario when the mobile user moves in a square multiple times. We intentionally set the size of each loop slightly bigger than the cache size, hence LRU always discards the page that will be visited next. However we can see that FAR works robustly here.

CacheSz(kb)	SquareSz	LRU	MRU	FAR
128	100×60	1867	291	336
256	200×60	1864	379	461

Table 6: Compare LRU, MRU and FAR When Mobile User Moving in Squares

From this set of experiments, we notice that FAR, which utilizes the semantic locality in terms of locations, is more effective in managing a semantic cache for LDD applications. It performs very well when the query workload exhibits a high degree of locality, and it behaves robustly under other workload types.

6. CONCLUSIONS AND FUTURE WORK

An LDD query workload is more likely to exhibit a semantic locality in terms of locations, rather than a static spatial locality defined by the fixed database physical organization. This unique characteristic makes semantic caching an ideal cache scheme for the LDD applications in mobile computing. In order to further utilize this intrinsic semantic locality among LDD queries, we developed the semantic cache replacement strategy FAR, which aims to let the cache contents move as the user moves. We believe that the semantic cache scheme with FAR is a very effective solution for LDD applications, which is also proved by our simulation study. Moreover, we examined the issues about how to build a model to abstract moving objects and formally defined LDD queries. We gave a try in LDD workload design too.

For future research, we intend to work along the following dimensions. First, we plan to design more types of LDD query workloads, and study the performance of semantic caching as well as FAR under them. Second, we will work on other cache management aspects in LDD applications, such as prefetching and cache admission policies. Meanwhile, we would like to apply the directional Manhattan distance approach of [5] to our semantic caching model and compare FAR with it. Moreover, we will also look at the other 2-dimensional space clustering approaches and analyze their impact on page caching.

7. ACKNOWLEDGMENTS

This work was supported partially by the National Science Foundation (NSF) under Grant No. IIS-9979458. The authors are very grateful to the anonymous reviewers for their valuable comments.

8. REFERENCES

[1] *CSIM18 Simulation Engine (C++ Version): User's Guide*. Mesquite Software Inc., Austin, TX, 1998.

[2] A. Acharya, B. R. Badrinath, T. Imielinski, and J. C. Navas. A www-based location-dependent information service for mobile clients. Technical report, Rutgers University, July 1995.

[3] B. Y. Chan, A. Si, and H. V. Leong. Cache management for mobile databases: Design and evaluation. In *Proceedings of ICDE*, pages 54–63, Orlando, Florida, February 1998.

[4] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In *Proceedings of EDBT*, pages 323–336, Cambridge, UK, March 1994.

[5] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivatava, and M. Tan. Semantic data caching and replacement. In *Proceedings of VLDB*, pages 330–341, Bombay, India, September 1996.

[6] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *Proceedings of ACM SIGMOD*, pages 259–270, Seattle, WA, June 1998.

[7] M. H. Dunham and A. Helal. Mobile computing and databases: Anything new? *SIGMOD Record*, 24(4):5–9, December 1995.

[8] M. H. Dunham and V. Kumar. Location dependent data and its management in mobile databases. In *Proceedings of DEXA Workshop*, pages 414–419, Vienna, Austria, August 1998.

[9] P. Godfrey and J. Gryz. Semantic query caching in heterogeneous databases. In *Proceedings of KRDB at VLDB*, pages 6.1–6.6, Athens, Greece, August 1997.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[11] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of ACM SIGMOD*, pages 332–342, Atlantic City, NJ, May 1990.

[12] B. T. Jonsson. Siumei: Design overview and class interfaces. Technical report, University of Maryland, 1996. Available from http://www.cs.umd.edu/projects/dimsum/.

[13] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(2):35–47, April 1996.

[14] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of PODS*, pages 261–272, Philadephia, PA, May 1999.

[15] K. C. K. Lee, H. V. Leong, and A. Si. Semantic query caching in a mobile environment. *Mobile Computing and Communications Review*, 3(2):28–36, April 1999.

[16] Q. Ren and M. Dunham. Semantic caching and query processing. Technical Report 98-CSE-4, Southern Methodist University, May 1998.

[17] Q. Ren and M. Dunham. Using clustering for effective management of a semantic cache in mobile computing. In *Proceedings of the International Workshop of MobiDE*, pages 94–101, Seattle, WA, August 1999.

[18] N. Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *ACM Transactions on Database Systems*, 16(3):535–563, September 1991.

[19] P. Scheuermann, J. Shim, and R. Vingralek. Watchman: A data warehouse intelligent cache manager. In *Proceedings of VLDB*, pages 51–62, Bombay, India, September 1996.

[20] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of ICDE*, pages 422–432, Birmingham, U.K., April 1997.