Cache Management for Mobile Databases: Design and Evaluation*

Boris Y. Chan Antonio Si Hong V. Leong
Department of Computing
The Hong Kong Polytechnic University
Hong Kong

Abstract

Communication between mobile clients and database servers in a mobile computing environment is via wireless channels with low bandwidth and low reliability. A mobile client could cache its frequently accessed database items into its local storage in order to improve performance of database queries and availability of database items for query processing during disconnection. In this paper, we describe a mobile caching mechanism for a mobile environment utilizing point-to-point communication paradigm. In particular, we investigate issues on caching granularity, coherence strategy, and replacement policy of mobile caching. Via a detail simulation model, we compare our proposed caching mechanism with conventional ones and discover that our mobile caching mechanism outperforms conventional ones in most situations.

1 Introduction

In a mobile environment, a set of database servers disseminates database information via wireless channels to multiple *mobile clients*. Depending on the affinity of individual database items, items of interest to most mobile clients should be broadcast from a database server to multiple clients while items of interest to single client should be disseminated over dedicated channels on demand [9]. Since a wireless channel suffers from a low bandwidth of 19.2 Kbps per channel and is also vulnerable to frequent disconnection, it is important to cache frequently accessed items (hot spot) into a mobile client's local storage. This improves the performance of database queries and the availability of database items for query processing during disconnection.

A caching mechanism is characterized by its caching granularity, cache coherence strategy, and cache replacement policy. Conventional caching usually requires a quite stable network, a reasonably high transmission bandwidth, and a high degree of locality among database items residing within a data page at the database server [6]. These conflict with the characteristics of a mobile environment.

In this paper, we investigate the above three issues of a caching mechanism in a mobile environment utilizing point-to-point communication paradigm, referred to as *mobile caching*. Caching mechanism for broadcast paradigm has been addressed in [1, 2, 8, 12]. The performance of mobile caching will be evaluated via a detail simulation model.

The remainder of this paper is organized as follows. In Section 2, we survey previous work on caching mechanisms. The design and implementation of various cache management issues for mobile caching are described in Section 3. In Section 4, we present the design of our simulation model. Section 5 presents some of our representative experimental results. Finally, we offer brief concluding remarks in Section 6.

2 Related Work

Caching mechanisms in conventional client-server environment are usually page-based [6], primarily because the overhead for transmitting one item or a page is similar in conventional client-server environment. Page-based caching mechanisms require a high degree of locality among the items within a page to be effective [4].

In practice, database items requested by different mobile clients via dedicated channels will differ much in a point-to-point mobile environment; otherwise the more effective and scalable broadcast paradigm should be employed to broadcast items of common interest [9, 15]. A physical organization that favors the locality exhibited by one client might result in poor locality for another. Database items within a page at a database server thus barely exhibit any degree of locality. Furthermore, mobile clients are powered by short-life batteries [9]. Caching a page will result in wasting of energy when the degree of locality is low. The overhead of transmitting a page over a low bandwidth wireless channel would be too expensive to be justified. It is, therefore, necessary to consider caching at a smaller granularity in this context.

Cached items will become out-dated when the *base items* (copies residing at the database server) are updated. A cache coherence strategy must be provided to update the cached items at each client. Conventional cache coherence strategies require the server to notify all relevant clients whenever an item is updated. Each mobile client, however, connects or disconnects from the wireless network freely and frequently. It is, therefore, not feasible for the server to keep track of all cached copies of individual items. A client should take a more active role in maintaining the coherence of its cached items and determining if a particular cached item should be invalidated.

*This research is supported in part by the Hong Kong Polytechnic University Research Grant number 350/570.

In the Leases file caching mechanism [7], each file, cached in the local storage of a client, is associated with a pre-specified *refresh time* which defines the duration within which the cached file could be regarded as valid in the client's local storage. When the refresh time expires, the client needs to contact the server for an updated file. It is, however, difficult to determine an appropriate refresh duration.

If a mobile client can provide unbounded disk storage, it can cache all database items accessed. However, the available storage for caching is often limited [13]. Furthermore, caching items that will barely be accessed will result in a waste of energy. A cache replacement policy is needed to retain only frequently accessed items for best performance.

In [5], various cache replacement policies for a conventional database system have been examined: optimal, WORST, least recently used (LRU), CLOCK, and least reference density (LRD). These policies are all page-based. In general, the performance of individual replacement policies is sensitive to the characteristics of queries initiated and the application environment. A general conclusion on the performance of the replacement policies cannot be recommended. In practice, the optimal policy is often approximated by LRU in conventional caching [6, 16]. LRU is further generalized into LRU-k [14] which identifies the replacement victim according to the time of the k^{th} previous access of a page. LRU is, thus, equivalent to LRU-1. In a mobile environment, since a client might change its location, the set of database items in which a client is interested might change over time as well. Therefore, we need to examine the suitability of conventional replacement policies in this changing access pattern and to develop other more suitable replacement policies for better performance.

Only until recently have caching mechanisms been investigated in a mobile database environment [2, 8]. In [2], an invalidation report is broadcast over a wireless channel to inform individual mobile clients about the invalidation of cached items. This requires a mobile client to keep tuning into the channel to invalidate and refresh its cached items. A disconnected client may miss the invalidation. In [8], an item is cached in a mobile client's local storage if the number of read operations performed on the item is greater than the number of write operations performed, as is usually the case. We believe that an item should be cached in the local storage if it is frequently accessed regardless of the access operations. Rather than disallowing an item to be cached if it is frequently updated, the caching mechanism should adapt to the situation.

3 The Design and Implementation of Mobile Caching

Mobile clients suffer from network disconnection quite easily and frequently. For instance, a recent experience in using a commercial wireless equipment to access our server data throughout our departmental building indicated that a client could be disconnected from the wireless network simply by isolating the client with several thick office partitions. Caching mechanism is needed to retain the frequently requested database items in a client's local storage. The more effective a caching mechanism in keeping the frequently accessed items, the better a query will perform and the more queries could be served during disconnection.

3.1 The Cache Model

We investigate three different levels of granularity of caching a database item in an object-oriented database (OODB), namely, ***attribute caching***, ***object caching***, and ***hybrid caching***. Intuitively, in attribute caching, frequently accessed attributes of database objects are cached in a client's local storage. In object caching, the objects themselves are cached. Finally, in hybrid caching, only frequently accessed attributes of those frequently accessed database objects are cached. This ensures that the cached attributes of the cached objects will have a high likelihood to be accessed in the future.

Several design issues need to be addressed in the implementation. First, a cache table is needed in each client to identify if a database item (attribute or object) is cached in local storage. Second, if a client is connected to a server, the client should be able to retrieve the cached items from the local storage and the uncached items from the server. The client will only retrieve the cached items otherwise. Third, an effective coherence strategy is needed to maintain the freshness of the cached items and finally, an effective replacement policy needs to be identified to retain the most frequently accessed items in a client's local storage.

3.1.1 The Cache Table

The cache table of a mobile client is simply a mini-database, which could be managed by its own object-oriented database management system. In its local storage, a class hierarchy rooted at class Remote and a class hierarchy rooted at class Cache are maintained. For each class X at the server, a corresponding class X is created as subclass of Remote and a class C_X is created as subclass of Cache. Remote is used to indicate if an item is cached while Cache is used to hold the cached value of an item. Collectively, the hierarchy rooted at Remote functions as a cache table.

For each attribute, a, of class X at the server, a user defined method with the same name, a, will be created for X in the client's local storage. The definition of method a will be described below. In addition, an attribute with name c_a is created for class C_X, with the same definition as attribute a in the server.

To illustrate, consider an **A***dvanced* **T***raveler* **I***nformation* **S***ystem* (*ATIS*) application [3] in which a group of tourists would like to retrieve places of attraction and accommodation information from a database server via a portable computer equipped with wireless communication interface. Figure 1*a* illustrates a simplified OODB for this traveler information system while Figure 1*b* indicates the (partial) structure of the local database maintained by a mobile client.

To cache attributes of an object, x, belonging to class X from the server, an object is created for X in

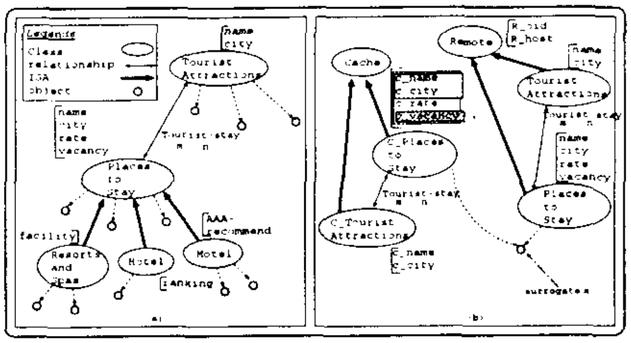

Figure 1: A sample ATIS database application

the client's local database. This local object is called a local *surrogate* for the remote object x at the server. Each local surrogate of X will inherit two attributes from Remote: R_oid holding the object identifier used by the server to reference x and R_host holding the address of the server where x originally resides. The local surrogate x is added to C_X via multiple membership modeling construct of OODB model. The surrogate, thus, inherits attributes defined for C_X, providing storages as placeholders to cache attributes of x. The value of an attribute, a, of an object, x, $a(x)$, will be cached under c_a of C_X. For instance, in Figure 1*b*, one object belonging to class Places to Stay is cached in the local storage of a client.

Each attribute of class X in a client's local database is a method. This method encapsulates the tasks involved in query processing. First, the initiated query is sent to the server. Second, for each attribute, a, required by the query, it retrieves $c\_a(x)$, for each local surrogate x of class X from the local storage if $a(x)$ has already been cached in $c\_a(x)$ of the local database. Third, it sends an *existent list* of (R_oid, a) to the server, informing the server about those attributes which have been satisfied locally, so that they do not need to be transmitted back. Finally, upon receiving such a list and evaluating the query, the server replies with a list of (`oid, attribute value`) pairs, for those that satisfy the query but are not cached in the client's storage. An advantage for encapsulating the cache model within a method is its transparency to a client. By simply having the method returning a null result during disconnection, the client can continue to operate using its locally cached items, without concerning if it is connected to the wireless network.

3.1.2 Attribute Caching

In attribute caching, after the server S has evaluated the query submitted by a client C, S only returns those attributes of those qualified objects that are requested by C. To illustrate, consider the following OQL query, Q, being evaluated against the ATIS database of Figure 1*a* by C:

```
select x.name, x.city
from x in Places to Stay
where x.vacancy > 0.
```

Assume that the local database schema at C is as shown in Figure 1*b*. Further assume that before the query is initiated, C's local database only contains surrogate x and only vacancy of x is cached as indicated by the shaded region (a) in Figure 1*b*. Finally, assume that only two objects, x and y, in S satisfy the query. When evaluating the where clause of the query, the method vacancy of Places to Stay will retrieve vacancy(x) from c_vacancy(x). A remote request will be sent to S containing the list: (R_oid(x), vacancy), as vacancy(x) has been cached. S will return a list of values for name(x), city(x), name(y), city(y), and vacancy(y) since only objects x and y satisfy the query. C can then, cache name and city, of surrogate x under c_name and c_city respectively, as indicated by shaded region (b) in Figure 1*b*. A new surrogate for object y will also be created in the local database. C can then cache name, city, and vacancy of y.

3.1.3 Object Caching

Each mobile client tends to have its own set of objects that it accesses most frequently. Furthermore, a client might access different attributes of an object in different queries it initiates. When the server receives a request, it might be worthwhile for the server to push all attributes of a qualified object, x, to the initiated client, thus eliminating future requests for x from the client. The client can cache the returned attributes under class C_X as in attribute caching. For instance, the shaded region (c) in Figure 1*b* shows that all attributes of x are cached when query Q is evaluated.

3.1.4 Hybrid Caching

In object caching, the database server will prefetch all attributes of a qualified object, x, to the client. It is very often that not all attributes of x will be accessed. This not only wastes the transmission bandwidth, but also occupies storage for caching other more frequently accessed attributes. Hybrid caching restricts the database server to prefetch only those attributes of a qualified object with a high likelihood to be accessed in the future. Only attributes with access probability above a *prefetching threshold*, ϵ, will be prefetched.

3.2 Cache Coherence

A cache coherence strategy usually involves cache invalidation and update schemes to invalidate and update an out-dated cached item. We note that most applications in a mobile environment will generate more read operations than write operations [8]. Furthermore, a mobile client usually can accept a slight degree of out-dated data in return for faster data retrieval. We propose a "lazy pull-based" invalidation approach in which each client is responsible to invalidate its cached item. In addition, we employ an on-demand update approach in which a stale cached item is only updated when it is next accessed. We illustrate the idea using object caching; coherence for attribute caching and hybrid caching will be similar.

We borrow the idea proposed in the *Leases* file caching mechanism [7] by estimating a *refresh time*

(RT) for each cached object. The estimation of the refresh time for an object depends on its update probability. If an object is updated frequently, its refresh time will be shorter. Consider an object, x, cached in a mobile client, C. The refresh time for x, RT_x, indicates the duration that x could be cached in a client, while remaining valid. In attribute and hybrid caching, since individual attributes of x are cached, the refresh time is associated with individual attributes. When C accesses x in response to a query, it checks the validity of x by determining if RT_x has expired. If RT_x has expired, x becomes stale. The existent list to the server will then not contain an entry for x. When the server returns x, in response to the request from C, it will estimate the refresh time for x and the new estimate will be sent along with x to C. In other words, the refresh time for an object is updated dynamically whenever the object is sent to a client from the server. This approach does not require a client to be always connected in order to invalidate x. Furthermore, if x is never accessed again, x will never be refreshed even after its refresh time expires.

To estimate the refresh time for each cached object, x, the inter-arrival duration of consecutive write operations, d_x, on x is maintained. The mean, $\overline{d_x}$, and the standard deviation, s_x, of the durations are computed. The refresh time of x is estimated as $\overline{d_x} + \beta_x s_x$. The value of β_x governs the frequency of refreshing x. It indicates the degree of deviation a client can tolerate on x. The smaller the value of β_x, the smaller is the refresh time and the higher the possibility that a client needs to request x when it accesses x in a query.

We define the notion of an *error* in accessing an object. Assume that a mobile client refreshes an object, x, at time t_1 and t_2. Between t_1 and t_2, the client might issue read operations on its local cached copy of x. For each read operation, r_x, initiated between t_1 and t_2, if the server performs a write operation, w_x, on x, before r_x, the value of x used by the client will be inconsistent with the actual value maintained at the server and hence, the read operation r_x results in an error. This definition is used in characterizing the performance versus coherence tradeoff in Section 5.

3.3 Cache Replacement

We consider a spectrum of replacement policies that adopt the access probabilities of database items as an indicator for the necessity of replacing a cached item. We illustrate the idea in the context of object caching; replacement policies under attribute caching and hybrid caching are similar.

For each object, a *replacement score* indicating the prediction of its access probability is estimated. The higher the score, the higher the estimated access probability, and the lower is its chance of being replaced.

The most straightforward way to compute the score for each object is by measuring the mean inter-operation arrival duration. The cached object with the highest mean arrival duration is replaced. We call this the *mean* scheme. Assume that the new duration for object x is $M_{x,n+1}$ and the estimated score for the previous n measures is $\overline{M}_{x,n}$. The new estimated score $\overline{M}_{x,n+1}$ can be computed incrementally as $(n\overline{M}_{x,n} + M_{x,n+1})/(n+1)$.

The mean scheme probably does not adapt well to changes in access patterns since every single trace from the beginning of the access history remains in effect. A better approach is to use a window for the statistical measures. Each object is associated with a window of size W, storing the access time of W most recent operations. The cached object with the highest mean arrival duration within the window is replaced. This is known as the *window* scheme, whose effectiveness depends on the window size, W. With a window size W_x, the new score $\overline{M}^{(W_x)}_{x,n+1}$ for object x is computed as $\overline{M}^{(W_x)}_{x,n} + (M_{x,n+1} - M_{x,n-W_x+1})/W_x$. A problem for the window scheme is the amount of storage needed in maintaining the W_x intermediate values.

To avoid the need of a moving window and to adapt quickly to changes in access patterns, our third scheme assigns weights to each arrival duration, such that recent durations have higher weights and the weights tail off as the durations become aged. The replacement score is the *Exponentially Weighted Moving Average* of arrival durations and is called the *EWMA* scheme. A parameter to EWMA is the weight, α, which ranges from 0 to 1. The current duration receives a weight of 1; the previous duration receives a weight of α; the next previous duration receives a weight of α^2 and so on. For object x with weight α_x, the estimated score for x, $\overline{M}^{(\alpha_x)}_{x,n+1}$, when adding a new measure, $M_{x,n+1}$, could be computed incrementally as $\alpha_x \overline{M}^{(\alpha_x)}_{x,n} + (1-\alpha_x)M_{x,n+1}$. The EWMA scheme is, in effect, similar to the LRD scheme described in [5]. We characterize the performance difference between LRD and EWMA in Section 5.

4 The Simulation Model

The experiments presented here are organized around three objectives. First, we would like to study the performance differences among attribute caching, object caching, and hybrid caching. Second, we would like to compare our replacement policies with conventional ones. Third, we would like to study the effectiveness of our coherence strategy in maintaining the freshness of cached database items.

The simulation model is implemented using CSIM. It consists of one OODB server and 10 mobile clients. Two channels, each having a typical wireless bandwidth of 19.2 Kbps, are shared among all clients to communicate with the server. One channel is used for upstream queries while the other is used for downstream results. The OODB has one class, Root, with 2000 objects, all residing in the server's disk initially. Each object contains 9 attributes of primitive-valued types and 3 one-to-one relationships to another object of Root. Each object has a size of 1024 bytes.

We model memory caching at both server and clients, in additional to storage caching at each client. LRU is employed for buffer management at the server and the clients since memory buffer replacement is implemented by the operating system. The bandwidth of disk is set to 40Mbps to model fast SCSI disk while that of memory is set to 100Mbps. The size of memory buffer at the server is 25% of that of the database, i.e., 500 objects. The size of storage cache at each mobile client is 20% of that of the database, i.e., 400 objects. The memory buffer at each client can hold 30 objects. We model a scenario in which the server has a large buffer compared with the database. The storage at each client is of a moderate size while its memory buffer is the smallest.

The size of a remote request and a reply message depends on the caching granularity, but both have an 11-byte header including an IP address and a CRC for error detection. Each query has a low selectivity of 1%, i.e., 20 objects, since the amount of information requested for each query in a mobile environment should be relatively small in general.

We experiment mobile caching along seven dimensions. Along the first dimension, we investigate the effect of caching granularity, $\mathcal{G}$ (attribute caching or AC, object caching or OC and hybrid caching or HC), on the performance of mobile caching as compared with no caching (NC).

Along the second dimension, we experiment with two heat distributions, $\mathcal{A}$, among the database objects. First, we adopt an 80/20 rule in defining the hot objects: 20% of the objects are picked randomly such that 80% of the accesses fall within this subset. This is called the *skewed heat* (*SH*) pattern. We ensure that the hot objects of each client are not identical. Second, we allow the 20% hot objects to be reselected after every $\mathcal{A}_C$ queries. This is called the *changing skewed heat* (*CSH*) pattern.

Along the third dimension, we experiment with two different types of query, $\mathcal{Q}$. In *associative query* (*AQ*), $\mathcal{Q}_a$ primitive-valued attributes of each selected object will be accessed. The attributes of each object selected by a query follow a uniform skewed distribution. All attributes have a non-zero access probability. In *navigational query* (*NQ*), for each selected object, $\mathcal{Q}_n$ inter-object relationship will be traversed in additional to accessing $\mathcal{Q}_a$ primitive-valued attributes of the object. When traversing the relationship, $\mathcal{Q}_a$ attributes of the related object will also be referenced.

Along the fourth dimension, we experiment with different replacement policies at a client's local storage, $\mathcal{R}_{disk}$. We experiment with LRU, i.e., LRU-1, LRU-k, LRD, mean, window, and EWMA schemes. We specifically select LRU, LRU-k, and LRD schemes for comparison with our schemes mainly because LRU is the most popular approach and LRD is intuitively similar to our EWMA scheme. LRU-k has also been shown in [14] to have a better performance than LRU when access patterns on objects change periodically.

We experiment the fifth dimension with two arrival patterns of queries, $\mathcal{P}$. With Poisson arrival, the mean arrival rate of queries is set to 0.01 [10]. With Bursty arrival, we model the vehicle traffic pattern [10], with the same average rate of 0.01 over one day. Of all the queries initiated within 24 hours, 80% of the queries are clustered within two bursty periods: from 07:00 to 10:00 (with mean arrival rate of 0.037), modeling the bursty traffic for traveling to work, and from 16:00 to 19:00 (with mean arrival rate of 0.027), modeling the bursty traffic for taking off from work. The two non-bursty periods will absorb 20% of the queries: the period from 10:00 to 16:00 has a mean arrival rate of 0.005 and the one from 19:00 to 07:00 has an even lower mean arrival rate of 0.0015, modeling working hours and happy/rest hours respectively.

The sixth dimension studies the effect of update probability of an object, $\mathcal{U}$, on the caching performance. For each object accessed by a query, there will be a probability $\mathcal{U}$ that it will be updated. All selected attributes of an object for update will be modified.

Finally, we experiment with the performance of a caching scheme during disconnection, since disconnected operation is an important aspect in a mobile environment [11]. Here, we would like to study the performance from two different perspectives. First, we vary the duration of disconnection of each client, $\mathcal{D}$. Second, we vary the number of clients that are disconnected, $\mathcal{N}$. Each experiment is conducted for a period of 4 simulated days, i.e., 96 simulated hours. The average metrics are measured. The standard deviation of our measurements is found to be very small, thus yielding very tight confidence intervals.

5 Performance Evaluation

We characterize the performance of a caching scheme by three metrics. Average cache *hit ratio* of all mobile clients measures the percentage of accesses that can be satisfied by retrieving a locally unexpired cached database item (attribute or object). Average *response time* of all mobile clients measures the average time spent (in seconds) from the moment a query is issued to the moment the results to the query are generated, either by using locally unexpired database items or remote results from the server. Finally, *error rate* measures the percentage of read errors the clients encountered (see Section 3.2). Six sets of experiments are conducted. The parameter settings of the experiments presented are summarized in Table 1.

Experiment		#1	#2	#3	#4	#5	#6
Parameter		Values for Each Experiment					
$\mathcal{G}$		NC, AC, OC, HC	HC			AC, OC, HC	
	ϵ	$\mu - 2 \cdot \sigma$					
$\mathcal{A}$		SH, CSH			CSH	SH	
	$\mathcal{A}_C$	500 queries			300, 500, 700 queries	N/A	
$\mathcal{Q}$		AQ, NQ			AQ		
	$\mathcal{Q}_a$	4					
	$\mathcal{Q}_n$	1			N/A		
$\mathcal{R}_{disk}$		EWMA	LRU, LRU-k, LRD, Mean, Window, EWMA	LRU, LRU-k, LRD, Mean, Window, EWMA	LRU, LRU-k, LRD, EWMA	EWMA	
	k	N/A	3			N/A	
	W	N/A	10		N/A		
	α	0.5					
$\mathcal{P}$		Poisson, Bursty			Poisson		
$\mathcal{U}$		0.1	0	0.1	0.1	0.1, 0.3, 0.5	0.1
	β	0	N/A	0	0	-1, 0, 1	0
$\mathcal{D}$		0					1-10 hours
$\mathcal{N}$		N/A					1, 3, 5, 7, 9

Table 1: Parameter settings for the experiments

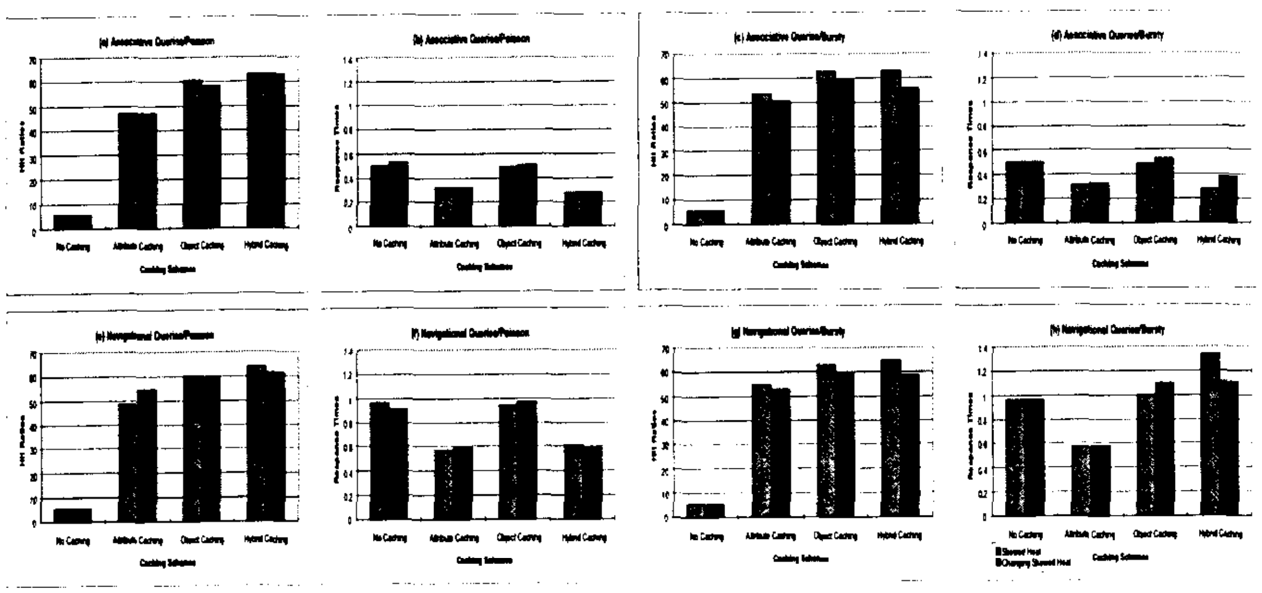

Figure 2: Performance of storage caching schemes *versus* no caching

5.1 Experiment #1

Our first set of experiments compares the performance of AC, OC, and HC with the base case, i.e., NC. In HC, the prefetching threshold, ϵ, is set to two standard deviations, σ, below the mean access rates of all attributes, μ. The base case is achieved by disabling storage caching at each client; only memory caching employing LRU at the server and each client is enabled. For storage caching, EWMA with $\alpha = 0.5$ (EWMA-0.5) is employed for $\mathcal{R}_{disk}$. The number of clients is fixed at 10, and update probability $\mathcal{U}$ is 0.1.

The results are depicted in Figure 2, arranged as a two-dimensional array of graphs. The first row (Figures 2*a* to 2*d*) illustrates the performance of AQ while the second row (Figures 2*e* to 2*h*) illustrates that of NQ. The first two columns show the performance of Poisson arrival pattern while the last two columns show that of Bursty arrival pattern. For the sake of clarity, only cache hit ratios and response times are depicted here. We will leave the error rates measurement to Experiment #5.

In Figure 2, it is clear that the base case performs a lot worse than any storage caching scheme. It has a much lower hit ratio and a much higher response time than those of any storage caching scheme. This is because a storage caching scheme trades network transmission for local disk access which has a much higher bandwidth than that of a wireless channel.

We observe that OC, in general, yields higher hit ratios than AC. This is mainly because OC will cache all attributes of the hot objects, thus eliminating extra requests when different attributes of the same object are requested in the future. It is interesting that despite the higher cache hit ratios from OC when compared with those from AC, OC results in higher response times as well. This anomaly is mainly due to "blind prefetching" in OC. Since OC prefetches all attributes of an object, the overhead for transmitting cold attributes of hot objects will be wasted. The overhead saved due to the high cache hit is offset by the even higher transmission delay due to the low bandwidth of wireless channel. *HC performs very well, achieving response times as low as those of AC and cache hit ratios close to those of OC.* Prefetching in HC appears to be "intelligent", in that most of the attributes prefetched will, most likely, be accessed in the future.

CSH access pattern results in slightly lower hit ratios and slightly higher response times than those of SH access pattern. This is because when there is a change of hot spot every 500 queries, the client needs to request new hot items from the server. The behavior is observed in both Poisson and Bursty query arrival patterns. *The only exception is the high response times from HC for NQ arriving in Bursty (Figure 2h).* We will explain this anomaly in Experiment #3 when we investigate HC in detail.

5.2 Experiment #2

We would like to identify the best possible performance of various replacement policies for storage caching. This occurs in a read-only environment, i.e., $\mathcal{U} = 0$, with only one client. We experiment with six replacement policies: LRU, LRU-k with $k = 3$ (LRU-3), LRD, Mean, window with $W = 10$ (Win-10), and EWMA with $\alpha = 0.5$ (EWMA-0.5). For LRD, the reference count of each database item is divided by 2 every 1000 seconds. Choosing 0.5 for α in EWMA has the closest intuitive meaning as dividing the reference count by 2. Only results of HC are depicted since it yields optimal performance in Experiment #1.

The results are illustrated in Figure 3. The first row depicts the performance of AQ while the second row depicts that of NQ. The first two columns show the performance of SH access pattern and the last two columns show that of CSH access pattern. Only cache

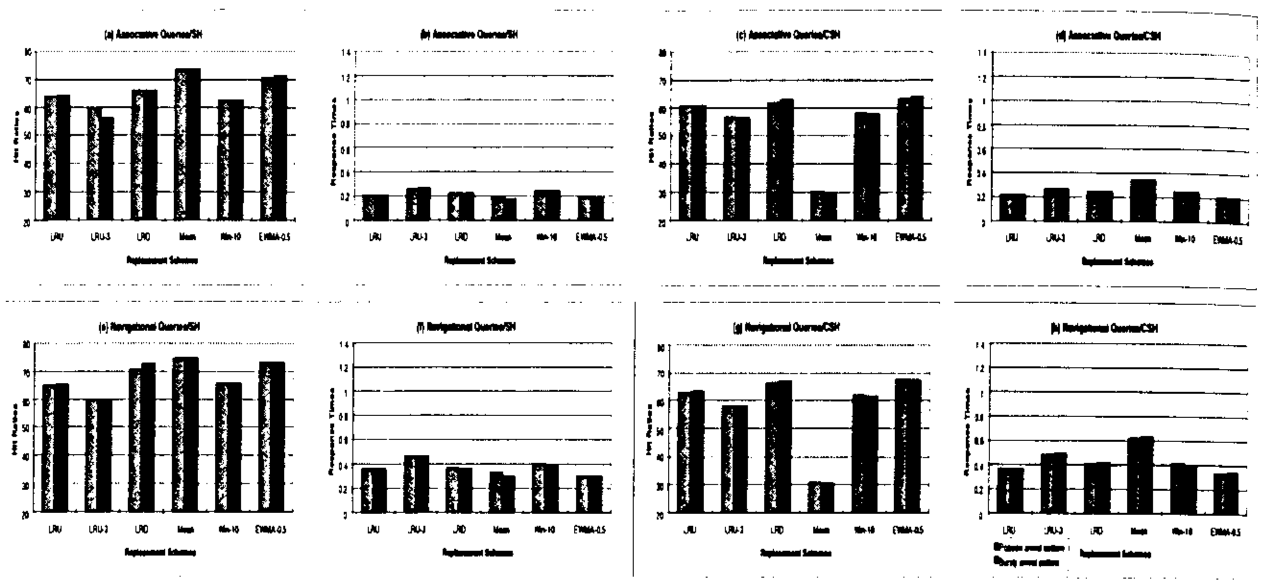

Figure 3: Performance of various replacement policies

hit ratios and response times are measured. There will be no error as there is no write operation.

For AQ, Figures 3*a* and 3*b* indicate respectively the cache hit ratios and response times for different replacement policies under SH access pattern. Here, LRU, LRU-3, LRD, and Win-10 perform similarly, resulting in a cache hit ratio in the range of 60%. Both Mean and EWMA-0.5 seem to be capable of capturing a larger portion of hot objects, yielding hit ratios in the range of 70%, as well as a lower response time. Both Poisson and Bursty arrival patterns exhibit similar behavior.

For NQ, Figures 3*e* and 3*f* depict the average cache hit ratios and response times for different replacement policies under SH access pattern. The interpretation and explanation of the figures are similar. We further observe that the response times are double those of AQ. This is because in NQ, for each object accessed in a query, $Q_n = 1$ additional object would also be referenced. This doubles the selectivity of each query.

For CSH access pattern, LRU, LRU-3, and Win-10 perform similarly. They are still able to maintain a cache hit ratio in the range of 50%. Mean performs a lot worse. Since Mean takes every access into account, it cannot adapt to the continuous changes in access pattern. LRD performs better due to its adaptive nature, about 5% higher hit ratios than those of LRU, LRU-3, or Win-10. EWMA-0.5 adapts even better, achieving another 5% higher hit ratios than LRD.

5.3 Experiment #3

We now compare the performance of various replacement policies for storage caching under a more realistic environment with write operations and multiple clients. We repeat Experiment #2 with identical settings except that $\mathcal{U} = 0.1$ and there are 10 mobile clients. Again, only the performance of HC is reported here. For the sake of clarity, only cache hit ratios and response times are depicted. We will leave the error rates measurement to Experiment #5. The results of this experiment are depicted in Figure 4, arranged in the same manner as Figure 3.

In the presence of write operations, the cache hit ratios drop considerably, up to 10% decrease which accounts to the increase in response times. This is because a query accessing an expired cached item will have to request the item from the server again. We further note that the response times from Bursty query arrival pattern are higher than those from Poisson arrival pattern. This is because for Bursty arrival pattern, queries arrive in a burst at the server. The results will be queued up at the downstream channel during bursty period since the low bandwidth wireless channel is not sufficient for delivering the results fast enough. This effect is especially serious for NQ since the selectivity is double that of AQ. One approach to address this problem is by a timeout heuristic. If the results of a query has been queued at the top of a queue for more than a duration threshold, the delivery of prefetched items for current query will be terminated. We will report more on the effect of this heuristic in the future.

From Experiments #2 and #3, we observe that mean is not stable. Its performance is very sensitive to changes in access patterns. LRU, LRU-3, LRD, and EWMA-0.5 perform better than Win-10. LRU performs slightly better than LRU-3 while LRD performs slightly better than LRU. EWMA-0.5, in turn, performs slightly better than LRD.

5.4 Experiment #4

We next compare the performance of LRU, LRU-3, LRD, and EWMA schemes since our previous two experiments have shown that these schemes perform similarly, within 10% difference, and relatively invariant to access patterns.

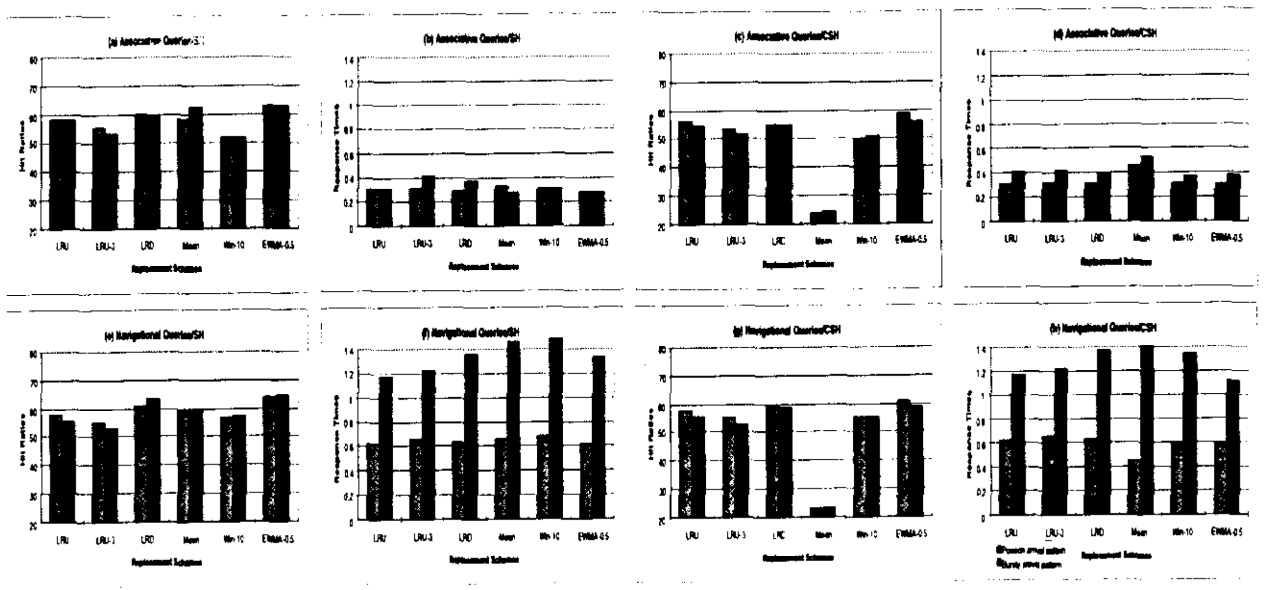

Figure 4: Performance of various replacement policies with write operations

We experimented these four schemes on CSH access pattern with changing rates of 300, 500, and 700 queries. Due to space limitation, only performance of AQ arriving in a Poisson pattern are depicted. The number of clients is fixed at 10 and update probability is fixed at 0.1. Again, we look at HC only. The results are depicted in Figure 5.

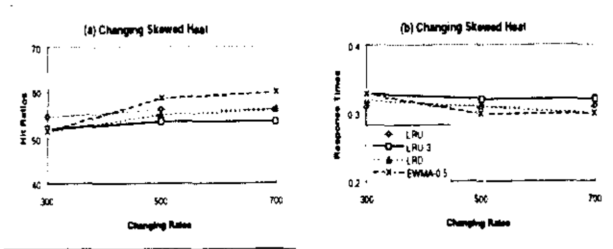

Figure 5: Comparison among LRU, LRU-3, LRD, and EWMA-0.5 schemes on CSH access pattern

Figure 5*a* illustrates cache hit ratios while Figure 5*b* illustrates the corresponding response times. LRU and LRU-3 *perform slightly better than the other* two schemes when the changing rate is fast, i.e., 300 queries. As the changing rate slows down, EWMA-0.5 and LRD start to outperform the other two schemes. EWMA-0.5 performs the best among the four schemes when changing rate is slower than 500 queries; as much as 7% higher cache hit ratios is recorded.

In CSH access pattern, a previously cold database item may become hot. Since LRU-based schemes will always cache the recently accessed item, they perform better for fast changing rate since the currently cached item will have a high probability of being hot. When the changing rate slows down, LRU-based schemes could not take advantage of the currently cached item and EWMA and LRD perform better.

So far, our experiments have shown that LRU performs slightly better than LRU-3. In [14], it was shown that LRU-3 will perform better than LRU if the access pattern of database queries exhibits a cyclic behavior, i.e., *the same set of database items are referenced by* database queries after a certain duration. In order to experiment this cyclic access pattern in our environment, we follow [14] in generating the set of database items to be accessed by each database query. The performance of LRU, LRU-3, LRD, and EWMA-0.5 is shown in Figure 6. Again, only performance of AQ arriving in Poisson pattern is depicted.

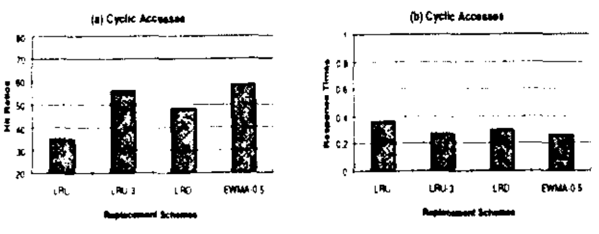

Figure 6: Comparison among LRU, LRU-3, LRD, and EWMA-0.5 schemes on cyclic access pattern

LRU suffers from this cyclic access pattern, incapable of retaining the useful database items. It is outperformed by LRU-3 with 21%. EWMA-0.5 also performs a lot better than LRD, by as much as 11%. Although not specifically design for the cyclic pattern, EWMA-0.5 is able to achieve reasonably well performance, close to that of LRU-3, which is designed for this cyclic pattern.

5.5 Experiment #5

Our fifth set of experiments investigates the effect of update probability on the performance. The error

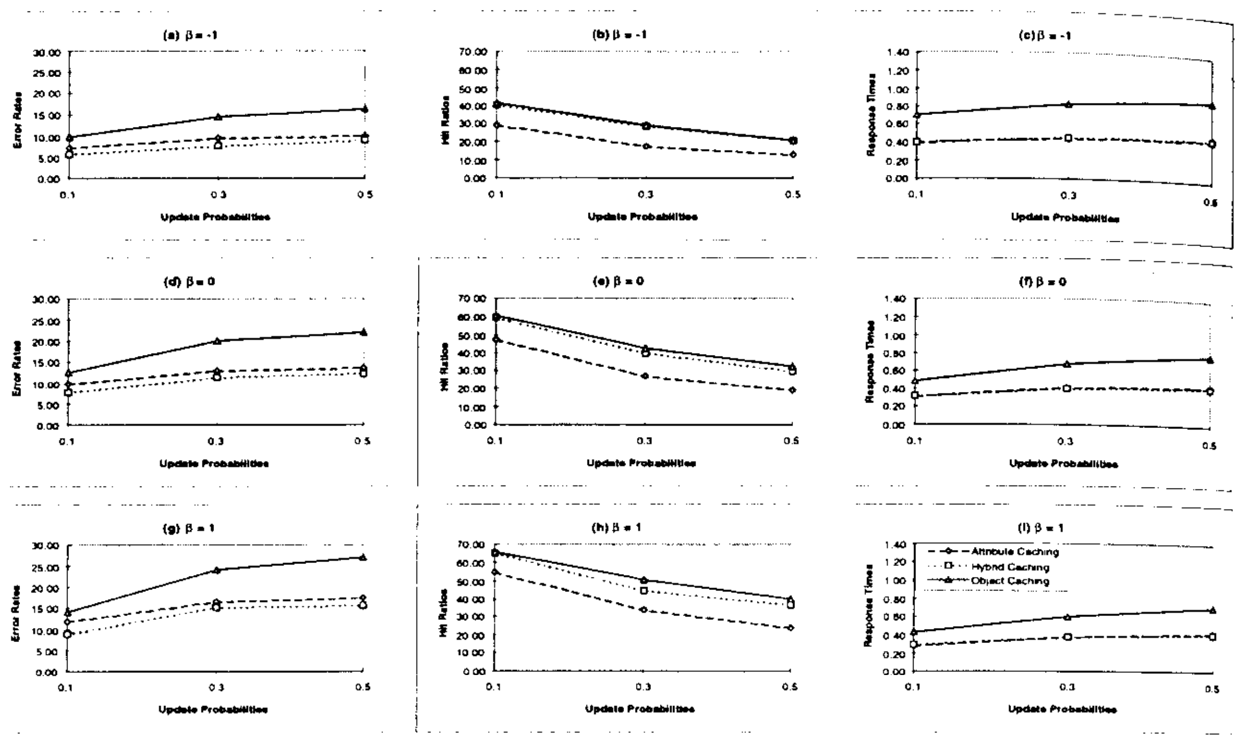

Figure 7: Error rates *versus* update probabilities

rates, hit ratios, and response times for AC, OC, and HC are measured. Due to space limitation, we only present the results for AQ with Poisson arrival pattern, operating on SH access pattern. with EWMA-0.5 replacement policy. The β value ranges from -1, 0, to 1 while update probability, U, ranges from 0.1, 0.3, to 0.5. We do not present the results for higher values of U since we believe that the number of write operations should be less than the number of read operations, especially in a mobile environment. Figure 7 depicts the results. The first row illustrates the measurements for $\beta = -1$, the second row for $\beta = 0$, and the third row for $\beta = 1$. The first column presents error rates, the second column hit ratios, and the third column response times.

We observe that OC, in general, has higher error rates than those of AC and HC. This could be explained as follows. Assume that a client reads attribute a of a cached object, x, at time t_1 and t_2 while attribute b of x is updated by another client at time t, $t_1 < t < t_2$, at the server. For OC, since the update of attribute b is an update on x, the object being read at t_2, the read operation at t_2 is an error. For AC and HC, the read operation at t_2 is not an error since attributes a and b are considered as different items.

AC and HC result in similar number of errors. However, HC can satisfy more read operations using locally cached items due to the higher hit ratios. Thus, the error rates of HC are slightly lower than those of AC.

The error rates increase with the update probability U. When U is low, there are few write operations. Since an error only occurs when there is a write operation followed by a read operation within any two consecutive refreshes, the error rate is low. When U increases, the probability that there is a read operation following a write operation within a refresh duration increases and the error rates increase accordingly. One might also observe that the error rates increase as β increases. This is because when β increases, the refresh time of a database item increases accordingly. This increases the probability of reading a stale item before the item is refreshed.

The hit ratios increase as β increases, since the refresh time of a database item increases with β, thus increasing the probability that a locally cached item could be accessed. In contrast, the response times decrease as β increases due to increased hit ratios.

5.6 Experiment #6

Our final set of experiments is to study the error rates during disconnection. The duration of a disconnection period, $\mathcal{D}$, for each client ranges from 1 to 10 hours. We vary the number of clients that are disconnected, $\mathcal{N}$, from 1, 3, 5, 7, to 9. The total number of clients is still fixed at 10. Figure 8 presents the error rates of AQ with Poisson arrival pattern, operating on SH access pattern, employing EWMA-0.5 replacement policy. The error rate is measured based on a "perfect" knowledge of all the events in the simulated system. Figures 8*a* to 8*c* depict the error rates for AC, OC, and HC respectively.

As shown in Figure 8, the error rates increase as the duration of disconnection increases in all schemes. This is because during disconnection, a client will con-

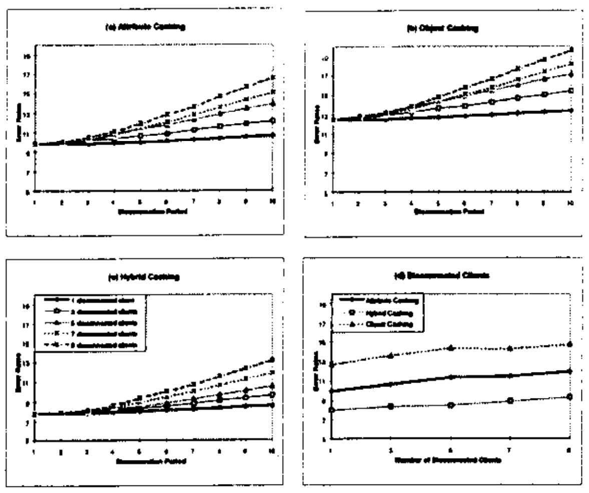

Figure 8: Error rates during disconnection

tinue to use its locally cached, but expired, items. As more clients are disconnected, the total number of errors will increase. The increase is relatively slow, however, as illustrated in Figure 8*d*, which shows the relationship between error rates and number of disconnected clients when the disconnected duration is fixed at 5 hours.

6 Conclusion

We have presented a framework for caching mechanism as one way to improve data access performance in a mobile environment. The caching mechanism is illustrated and implemented on object-oriented database model. We have shown that page-based caching is not suitable in this mobile context and proposed three different caching granularities, namely, attribute caching, object caching, and hybrid caching. We have also shown that conventional cache coherence and replacement schemes are not as effective and modified strategies which adapt to object access patterns have been proposed. The behavior of our caching mechanism is illustrated through a series of simulated experiments.

We intend to extend this study in several directions. First, this study assumes that each mobile client only communicates with one server. In real applications, a mobile client might request items from multiple servers, possibly under different cells. This further complicates the problem because the contact server for a client might have to request and even cache items from other remote servers on behalf of the client which initiates the query. Second, we would like to investigate into the correctness conditions on concurrent query processing. Since mobile clients are highly dynamic and autonomous, the notion of serializability for conventional transaction processing would be too restrictive for efficient concurrency control in this context. Relaxed notions should be explored. Finally, we are incorporating our results into our prototype to be further validated with the simulated results.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *Proceedings of the ACM SIGMOD*, pages 199–210, 1995.

[2] D. Barbara and T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In *Proceedings of the ACM SIGMOD*, pages 1–12, 1994.

[3] M. Choy, M. Kwan, and H.V. Leong. On Real-time Distributed Geographical Database Systems. In 27^{th} *Hawaii International Conference on System Sciences*, pages 337–346, 1994.

[4] D. DeWitt and D. Maier. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *Proceedings of VLDB*, pages 107–121, 1990.

[5] W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, pages 560-595, December 1984.

[6] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *Proceedings of VLDB*, pages 596–609, 1992.

[7] C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of SOSP*, pages 202–210, 1989.

[8] Y. Huang, P. Sistla, and O. Wolfson. Data Replication for Mobile Computers. In *Proceedings of the ACM SIGMOD*, pages 13–24, 1994.

[9] T. Imielinski and B. Badrinath. Mobile Wireless Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):18–28, 1994.

[10] J. Jannink, D. Lam, N. Shivakumar, J. Widom, and D.C. Cox. Data Management for User Profiles in Wireless Communications Systems. Technical report, Computer Science & Electrical Engineering Department, Stanford University, 1994.

[11] J.J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of SOSP*, pages 213–225, 1991.

[12] H.V. Leong and A. Si. Database Caching over the Air-Storage. *The Computer Journal*. To appear.

[13] C. Min, M. Chen, and N. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Proceedings of International Conference on Extending Database Technology*, pages 323–336, 1994.

[14] E. O'Neil, P. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the ACM SIGMOD*, pages 297–306, 1993.

[15] A. Si and H.V. Leong. Query Processing and Optimization for Broadcast Database. In *Proceedings of International Conference on Database and Expert Systems Applications*, pages 899–914, 1996.

[16] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 1996.