Communication-Efficient Query Answering with Quality Guarantees in Client-Server Applications

Michal Shmueli-Scheuer
UC Irvine, USA
mshmueli@ics.uci.edu

Amitabh Chaudhary
University of Notre Dame, USA
achaudha@cse.nd.edu

Avigdor Gal
Technion, Israel
avigal@ie.technion.ac.il

Chen Li
UC Irvine, USA
chenli@ics.uci.edu

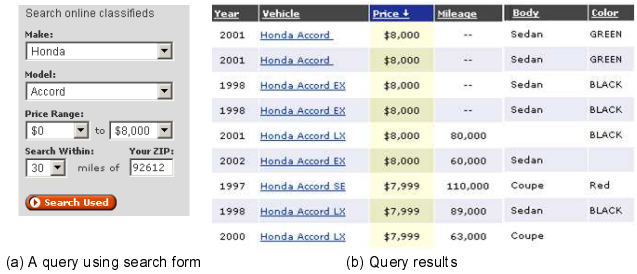
ABSTRACT

We study how to reduce costs in client-server web based applications with dynamic data on the server. Client-side caching can help mitigate costs because the client can use the cached data to answer queries. Allowing some tolerance on the data staleness to answer queries makes it possible to significantly reduce costs. For example, if the user can tolerate data that was received 2 hours ago, we can use the cached data to provide the answer and to save some costs. In this paper we present useful algorithms under different cost models, we provide 2-approximation offline algorithm, as well as (k+1) competitive online algorithm and family of Heuristics. We validate our methods through extensive experiments.

1. INTRODUCTION

In a client-server database environment, a client issues queries to the data residing on a server. Such an environment exists in many emerging applications, especially on the Web. As an example, assume we want to build a mediation system [13] that integrates car information from various data sources. The system has a mediator that accepts user queries (e.g., *"find Honda Accord cars priced less than $8000, and 30 miles within the zip code 92617"*). The mediator answers such a query by sending corresponding queries to the underlying sources, collecting the answers, and possibly postprocessing the results. Assume one of the data sources is cars.com, an online car buying and selling service. Figure 1(a) shows a submitted query using a Web form of the server, and Figure 1(b) shows part of the query results, which are a list of cars. Clearly client users want fast and accurate query results even in the face of rapidly changing data. In this example, the cars.com Web site can be viewed as a database server with car information, and the mediator is a client that issues queries to the data on the server.

We study how to reduce communication costs in such an environment. Costs can be in terms of the number of queries submitted to the server, the amount of data transferred over the network, or the overhead on the server. There are many reasons we need to reduce these costs, e.g., we want to reduce the query workload on the server, and we may have limited network bandwidth. Client-side caching can be used to alleviate the costs. That is, the client can cache the results of earlier queries, which can be used to answer future queries locally, or by sending the server a slightly modified, more efficient query.



(a) A query using search form (b) Query results

Figure 1: A query to cars.com and part of the query results.

We can further reduce the costs if the queries can tolerate some data that is not up-to-date. For instance, suppose the query results in Figure 1(a) are cached at the client, and a new query asks for similar cars except that the price is between $5000 and $9000. If the client knew that the new query can tolerate some staleness in the answer, for example: the *age* of the data (formally defined in Section 3.1), then the client can just ask the server the corresponding cars within $8000 and $9000. Such a decision depends on the freshness requirement on the answer to the query.

As illustrated by this example, many client-server applications have the following four characteristics.

C1: The data on the server is dynamic. New cars can become available at the server, existing cars can be sold (deleted), and prices of existing cars can change.

C2: Each client query is answered using results with a predefined quality requirement, while the quality can be defined in various ways.

C3: The client can store as much data as necessary, even all the data on the server. This is increasingly true with lowering storage costs. The main computing bottleneck is the limited communication resources between the client and the server.

C4: The client and the server communicate with each other on a query basis (as opposed to communicating on an object basis): the client issues a query with semantic conditions, and

Proceedings of the 10th International Workshop on Web and Databases (WebDB 2007), June 15, 2007, Beijing, China

the server returns objects satisfying these conditions.

In this paper we study research challenges on reducing costs in such applications, where different queries have different requirements on how fresh their answers should be.

In this work we make the following contributions.

• We present the problem setting, include issues such as how to define quality of query answers, different communications cost models (Section 3).

• We develop offline and online algorithms for reducing communication costs to answer queries to meet their freshness requirement (Section 4). We present family of efficient and practical heuristics (Section 5).

• We conduct intensive experiments to evaluate the algorithms (Section 6).

2. RELATED WORK

There has been intensive work in the literature on reducing communication costs in client-server applications. It is not our intention to compare our setting with all of them. Here we briefly compare ours with some of the representative works in terms of their differences on the four characteristics. Dar et. al [5] proposed the concept of "semantic data caching," in which the client maintains a semantic description of the data in its cache, and decides whether it should contact the server for data not in the cache. The data needed from the server is specified as a remainder query. An example of a semantic description is "cars made between 1998 and 2000 and priced between $3000 and $5000." Our setting differs from theirs on characteristics **C1**, **C2**, and **C3**. They assumed that the client has limited storage space, and thus they focused on cache replacement policies, without considering data updates on the server. In addition, they did not consider quality of query answers. We consider the case where the client site has enough cache to store data, the data can change, and the user may tolerate some staleness on the requested data.

Keller and Basu [9] proposed predicate-based caching, in which the client also uses possibly overlapping query-based predicates to describe the cached data on the client. These predicates are similar to semantic descriptions used in [5]. In the work [9], the server is collaborative in the sense that it is responsible for notifying the client about data updates satisfying these predicates. Our setting differs from this earlier work mainly on characteristics **C2** and **C3**. We consider query quality guarantees defined using different measures, which is not a focus of their work. In addition, they studied cache-replacement policies, while we assume the client has enough storage space. Furthermore, we believe more research and experimental studies are needed to consider a variety of communication models.

Many stale caching studies have been proposed recently [3, 2, 11, 1]. Their settings are different from each other with respect to the four characteristics. For instance, the work in [11] assumes that the server is collaborative and can push updates to the client. The works in [3, 2, 1] assume that the server is not collaborative and cannot push data updates. [3, 2] consider a given query workload, and try to decide a synchronization policy to maximize some freshness measure under a limited bandwidth. [1] consider a tradeoff between latency and recency. The main difference between our setting and these works is on characteristics **C1** and **C4**. We assume that the server can have deletions and insertions of objects, while these earlier works assume a static set of objects (e.g., Web pages that need to be crawled). In addition, we assume the client and the server exchange queries and their results, while these earlier works assume that both sites communicate with each other on an object basis.

3. FRAMEWORK

Consider a client-server environment, where the data on the server is stored in a relational table. Each record in the relation represents an object, such as a car, a book, a restaurant, or a house. A client issues a query to the server, which specifies conditions on attributes of the table, and asks for the records (objects) satisfying these conditions. As an example, consider a client-server environment where the server has car information stored in a relation `car(make, model, year, mileage, color, price)`. A query $Q1$ asks for Honda Accord cars priced between $3000 and $8000.

Each client query comes with a *quality requirement*, either specified explicitly by the user, or provided implicitly by the client. This requirement indicates the tolerance of the query to accept answers that might not be up-to-date. The system makes sure that the answers to the query meet the requirement. Such quality guarantees can satisfy various application needs, and provide good opportunities for the system to optimize queries sent to the server in order to reduce the costs. Section 3.1 presents the *elapsed time* quality measurement and discusses several quality aspects that need to be considered when developing techniques in this setting.

In these kinds of applications, the server is typically non-collaborative and the client cannot rely on any notification from the server about changes in the data. Using the *elapsed time* measurement we are still able to provide some guarantee to the quality of the data. To provide this guarantee, we assume an updates model of the objects in the database, similarly to [7]. Section 3.2 discusses the costs, related with pulling the server.

3.1 Quality Measures

Each query Q is associated with a quality requirement $\tau(Q)$ that needs to be satisfied. In this paper we measure quality in terms of *elapsed time* as in [4]. Formally, the quality of an object o at time t at the client is defined as $t - t'$, where t' is the last time the object was retrieved from the server. To enforce a quality requirement using this measurement, the client does not require the server to be collaborative, since the measure does not depend on the time the object was last updated.

Given a set of objects as an answer to a query Q, we define the quality requirement $\tau(Q)$ using the quality of the objects in the answer. Let $ANS_c(Q)$ be the set of objects (at the client side) satisfying the query conditions. For each object in $ANS_c(Q)$ we maintain the age of each object(last time it was retrieved). Thus, for query Q, we define $age(Q)$ to be the age of the oldest object in $ANS_c(Q)$. For query Q, a quality requirement will be satisfied if $age(Q) \leq \tau(Q)$.

Figure 2 shows a representative workload of our problem consisting of 4 range queries and their arrival time, each with tolerance of $\tau(Q_i) = 5$ time units. For example, query $q2$, arriving at time $t = 5$, allows tolerance of 5 time units and asks about objects with price ranging from $10K$ to $40K$. The number of objects in the price ranges $[10K,20K]$,$[20K,40K]$ and $[40K,50K]$ are 4,6 and 5 correspondingly.

3.2 Cost Measures

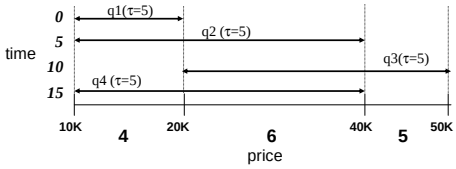


Figure 2: range queries with their arrival time and quality requirement.

We consider a cost model of the form

$$\alpha + \beta \times N \quad (1)$$

where α captures costs such as latency, β captures the bandwidth cost, and N is the size of the transferred data. The most realistic cost model is a generalized model, which considered both overheads and data size.

In this work we will study the general model of Eq. 1, as well as two simpler variations of it. The first is a simple communication cost model, where each interaction between the client and the server has a fixed cost ($\alpha \neq 0$ and $\beta = 0$). This is a realistic model for systems in which the communication cost is mainly determined by the number of messages between the two sides. We assume that there are no restrictions imposed by servers for client connections (e.g., politeness constraints [6]). Under this model, our goal is to minimize the number of times the client contacts the server.

For applications where the size of the transmitted data is important, we set $\alpha = 0$ and $\beta \neq 0$. With this setting, our goal is to minimize the size of the transmitted data.

3.3 Optimization Problem

Consider a numeric attribute where each query has a *range condition* for this attribute, and asks for information about objects satisfying the range condition. In addition, each query specifies its *quality requirement*. For a given queries workload and updates model, we want to decide the times in which each query (or sub-query) should contact the server in order to minimize the communication cost, subject to the quality requirement.

4. ALGORITHM FOR DATA PRE-FETCHING

In this section we describe and analyze algorithms for the problem under the various cost models. We give optimal algorithms for the simple models. For the general cost model we introduce a provably 2-approximate offline algorithm. Finally, we discuss an online scenario, and construct an algorithm with a competitive ratio roughly bounded by, what we call, the *cost ratio*. This is significant since we show that every online algorithm has a competitive ratio that is at least half the cost ratio.

Due to space considerations, we refrain from presenting some of the algorithms in detail and refer the interested reader to the online version of the paper.[1] There, proofs to complexity and correctness propositions can be found as well.

4.1 Optimal Greedy Algorithms

For the simple cost models we describe optimal greedy algorithms with an example and present their complexity.

4.1.1 Costs as Overhead ($\alpha \neq 0, \beta = 0$)

[1]http://www.ics.uci.edu/ mshmueli/webDBfull.pdf

It is easy to show that a greedy algorithm (denoted ***GreedyA***) which contacts the server when the quality requirement for a query is not satisfied and asks for all existing objects, generates an optimal plan. The complexity of such an algorithm is $O(n \log n)$ [8] where n is the number of queries. Figure 3(a), shows the times and the sub-queries for the example from figure 2. For simplicity we assume that each contact with the server costs 10, and that there are no insertions or deletions, i.e., only updates are considered. For ease of exposition, we assume that between client contacts, data at the server has already been completely changed. Recalling that we want to minimize the total communication cost, the client should contact the server at times $t = 0$ and $t = 10$. At each contact the client retrieves all the objects from the server (note that cost is only determined by the number of contacts, independent of the size of the downloaded data). Therefore, the total cost for this plan is 20.

4.1.2 Costs as Data Size ($\alpha = 0, \beta \neq 0$)

It can be shown that a greedy algorithm asking only for the needed objects each time it contacts the server, yields an optimal solution. The complexity of such an algorithm is $O(n \log n)$ [8] where n is the number of queries. We denote this greedy algorithm, ***GreedyB***. For example, Figure 3(b) illustrates the times and the corresponding sub-queries. Assuming the cost of transmission per object is 2, and again, that we only have updates (price attribute at the server has already been changed) the total cost is 50 (4 at time $t = 0$, 6 at $t = 5$, 5 at $t = 10$ and 10 at $t = 15$).

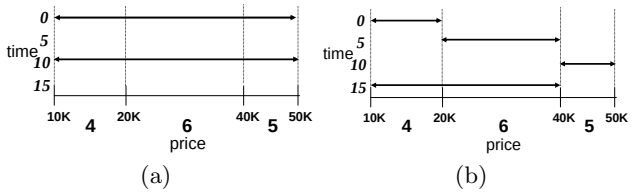


Figure 3: Running Greedy Algorithms on the example (a) *GreedyA*-optimal plan, total cost of 20(b) *GreedyB*-optimal plan, total cost of 50

Next we show how to generalize these two approaches into one that consider both the overheads and the data size.

4.2 Generalized Cost Model- Offline Algorithm

We first consider the offline case in which we assume that the client has full knowledge of the queries, their arrival times, the client tolerance, and the range these queries are interested in. Recall that the output of this algorithm should be the time instances when the client needs to contact the server and the corresponding sub-queries for each time instance. First let us describe the main idea of the offline algorithm.

The algorithm combines algorithm ***GreedyA*** and ***GreedyB*** in the following manner: the contact time instances will be derived from algorithm ***GreedyA*** and the sub-queries from ***GreedyB***. For each sub-query that both greedy algorithms agree on (i.e., fired at the same time instance), it will add the sub-query to the query for that instance. For those sub-queries that the two greedy algorithms do not agree on, it will replace the sub-queries with two queries, one will be added to the "pervious" contact time instance and another to the "next" contact time. (Note, if any such query has already

been chosen in a previous step, that query is skipped.)

Figure 4 shows how to construct the offline plan. For example, the algorithms agree on {time $t = 0$, sub-query [10K,20K]}; thus, we add this sub-query to the plan (depicted by the solid arrow). However, for {time $t = 5$, sub-query [20K,40K]} they do not agree; thus, we duplicate the sub-query and we add it to the previous contact time, $t = 0$, and to the next contact time, $t = 10$ (depicted by the dashed arrow). Note that for {time $t = 15$, sub-query [20K,40K]} we should duplicate the sub-query and add it to time $t = 10$ and for future time (which does not appear in the figure). However, since we have already added this sub-query to the plan at $t = 10$ (because of a previous query), we do not need to add it again. Following the same idea, we eventually generate the whole plan which appears at the right most side of figure 4.

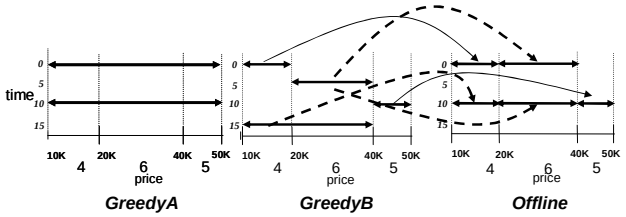


Figure 4: Constructing 2-approximate plan, with total cost=70

THEOREM 4.1. *The greedy algorithm under the general cost model is 2-approximate.*

4.3 Online Algorithm

We now describe an online algorithm for the general cost model with a bounded competitive ratio and a nearly matching lower bound on the competitive ratio of any deterministic online algorithm in this model. We call the algorithm Ratio; it is based on what we shall define as the *cost ratio* of the network.

Let us normalize the cost parameters such that the size of the smallest query is 1 and the size of the total data (data in the entire range) is L. Thus, the cost of the smallest possible query is $\alpha + \beta \cdot 1$ and of the largest possible query is $\alpha + \beta \cdot L$. The cost ratio of a model, denoted k, is defined :

$$k = \min\left\{\frac{\alpha}{\beta}, \frac{L \cdot \beta}{\alpha}\right\}.$$

Intuitively, it is the minimum of two ratios: (1) the latency cost to the minimum bandwidth cost, and (2) the maximum bandwidth cost to the latency cost. For a given amount of data L, the maximum possible value of the cost ratio is $\sqrt{L}$ (when $\alpha/\beta = \sqrt{L}$). Ratio is described in Figure 5. It simply chooses between GreedyA and GreedyBbased on the cost ratio. We can bound the competitive ratio of Ratio as stated below.

Ratio(α, β, L)
  If($\alpha/\beta \geq L\beta/\alpha$)
    Use GreedyA to process σ.
  Else
    Use GreedyB to process σ.

Figure 5: The online algorithm Ratio for the latency-bandwidth model.

THEOREM 4.2. *Algorithm* Ratio *for the general cost model is* $(k+1)$*-competitive, where* k *is the cost ratio.*

This simple algorithm is actually nearly optimal—no deterministic algorithm can perform much better than Ratio, as stated below.

THEOREM 4.3. *Every deterministic online algorithm for the general cost model has a competitive ratio at least* $(k+1)/2$ *where* k *is the cost ratio.*

5. HEURISTICS-BASED APPROACHES

In many cases we need more efficient ways to find the best solution. The offline algorithm assumes knowledge of the entire sequence in advance, which is unrealistic in web-applications. The online does not have such requirements. It, however, is pessimistic, concerned as it is about the worst case and designed to optimize for worst-case sequences, which occur infrequently in real-world applications. Any algorithm that assumes that the sequence will not turn out to be one of the worst-case ones, can makes "risky" decisions that are better than the online if the sequence is not worst-case. Such an algorithm can perform better on practical or realistic input sequences than the online algorithm. We now present heuristics that are designed to be adaptive as well as practical.

In the following heuristics, we take into consideration the most recent history. We define a window of size W, for which we maintain some statistics that help us decide whether we should pre-fetch some data. Whenever a contact with the server is issued, we need to consider pre-fetching of other data items. This pre-fetching (and caching) can save us additional contacts to the server later on.

5.1 Query-based Heuristic

The main idea behind this heuristic is that queries interested in similar objects may also have similar quality requirements. Thus, for each query in the window we maintain an average tolerance of the query instances denoted $AVG(Q)$. For example, if a query Q is: cars with price in [16K,18K] and two instances Q' and $Q''$of Q arrive with corresponding tolerances $\tau(Q') = 10$ minutes and $\tau(Q'') = 20$ minutes, then the average tolerance of this query is $AVG(Q) = 15$ minutes. In addition, for each query Q, we monitor the $age(Q)$ of the query to determine the staleness of the data in the cache in order to answer the query (as defined in 3.1). The pre-fetch rule is:

$$age(Q) > AVG(Q) \rightarrow prefetch(Q) \qquad (2)$$

5.2 Interval-based Heuristic

The attribute domain (for example, the car's price) can be divided into intervals. The intervals are formed by the starting and ending points of the ranges in the query condition. For each interval i, we compute its average tolerance value (denoted $AVG(i)$) to be the average tolerance of all the queries in the window that overlap the interval:

$$AVG(i) = \frac{\sum Q(\tau)_{\text{Q overlap interval i}}}{\sum j_{\text{j=1:Q overlap interval i}}}$$

For example, if Q: car price between [16K,18K] and Q': car price between [17K,20K] with corresponding tolerances $\tau(Q) = 10$ minutes and $\tau(Q') = 20$ minutes, and assuming that these are the only queries that overlap interval [17K,18K],

then the average tolerance is $AVG(i) = 15$ minutes. The prefetch rule is

$$age(i) > AVG(i) \rightarrow prefetch(i) \quad (3)$$

where $age(i)$ is the age of the oldest object in interval i.

6. EXPERIMENTAL RESULTS

In this section, we present experimental results that demonstrate the usefulness of data caching and pre-fetching in reducing the communication costs, and compare the performance of the various approaches we presented in the earlier sections for different scenarios. We used synthetic workloads generated with controlled variations in number of queries, number of objects, query lengths, etc., as follows. Both the objects and the queries were assumed to form clusters following a Gaussian distribution, the mean and variance of which were chosen uniformly at random. We used well-known techniques [12, 10] to generate the length of a query as a Gaussian distribution with different means and variances according to the domain size. The arrival times of queries was generated using exponential distribution. For each setting, we ran the experiments on 100 different workloads and computed the average (results were very stable).

Using these workloads we tested our methods in simulated client-server environments. We measured the costs, considering both the overhead and the data costs. All our experiments were implemented in C compiled using visual C compiler. We ran the experiments on a machine with Pentium M with 1.6 Ghz processors with 1 GB cache, and Windows operating system. In the following we present our results and give analytical explanations for the differing behavior of the algorithms.

6.1 Offline Vs. Online

The offline algorithm is a 2-approximation, so in the worst case its cost is twice the optimal algorithm. The online algorithm is a $(k+1)$ competitive, so in its worst case the cost will be $(k+1)$ time the optimal. Thus, it is possible that for different sequences the offline will perform better then the online and vice versa. Figures 6(a) and (b) show the communication cost as a function of time. Figure 6(a) illustrates a case where the offline outperforms the online (***GreedyA***), while Figure 6(b) shows the opposite case in which online (***GreedyB***) is better then offline.

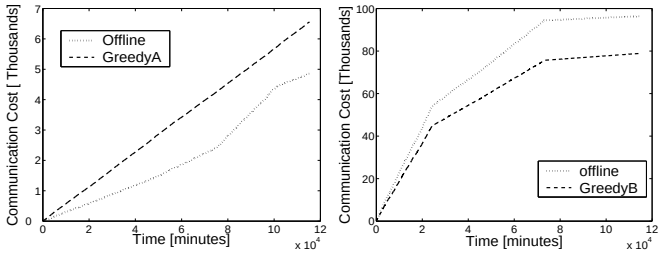


Figure 6: 10000 **queries,** 30000 **updates,** 80 **days (a)** $\alpha = 1, \beta = 0.01, L = 100$**(b)** $\alpha = 1, \beta = 0.2, L = 200$

In order to evaluate the performance of the heuristics vs. the online algorithm, we must overcome this flip-flop phenomena and define a fixed baseline (denoted *Quasi-baseline*). Such that, for each time t, the baseline cost is as follow:

$$cost\_quasi(t) = max\left\{\frac{cost\_online(t)}{k+1}, \frac{cost\_offline(t)}{2}\right\} \quad (4)$$

The *Quasi-baseline* will give a lower bound on the optimal cost and will be used as the baseline for comparisons later in Section 6.3.

6.2 Heuristics Evaluation

The heuristics are independent of specific values of α and β. Thus, we can evaluate their behavior with respect to other parameters such as query length and tolerance. In the first set of experiments we varied the query's length. The range attribute values were between 1,000 and 30,000. The number of queries were 10,000 for a period of 21 days, with tolerance (uniform) at random of 24 minutes. We set the history window to W=2 days. We let the query lengths vary from 200 to 4,000. Table 1 shows the number of connections and the number of transferred data for the heuristics. For small degree of overlapping (e.g., length=200), each query covers a unique interval and thus the heuristics converge to a similar solution. However, when the query lengths increased such that each query overlapped with more intervals, the query-based heuristic consistently contacts the server fewer times then the interval-based but each time transfer more objects. This is due to the fact that the interval-based heuristic corresponds to sending the remainder queries while the query-based approach sends whole queries.

query length	query-based		interval-based	
	Connections	Objects	Connections	Objects
200	1522	579800	1524	576702
400	1548	1028792	1686	924578
600	1503	1694400	1749	1638864
1000	1524	2770169	1699	2678217
2000	1406	4262425	1995	3480080
4000	1458	7119728	1851	5728601

Table 1: Number of connections and Objects for query and interval based heuristics

To evaluate the goodness of the heuristics we compare them to the baseline of the offline solution which is also independent of specific values of α and β. Figure 7 shows the number of connections and number of transferred objects with respect to the offline algorithm. For example, for query length of 1000, the number of connections for query-based was 1.83 and interval-based was 2.04 times that of the offline. However, the number of transferred objects were 0.92 and 0.89 respectively.

To verify the results, we conducted another set of simulations in which we varied the tolerance of the queries. The range attribute values were between 1,000 and 30,000. The number of queries were 10,000 in 21 days, the history window was W=2 days and the queries overlap was 1,000 on average. Figure 8 shows the number of connections and transferred objects for the offline, interval, and query based heuristics. For all cases, we can see that the results are consistent with our observation for the behavior of the interval and query based heuristics (the query-based is always **left** (less connections) and **up** (more data) comparing to the interval-based). The figure also show the number of connections and transferred objects of the heuristics compare to the offline. For instance, when tolerance=30, the number of connections for query-based was 1.91 and interval-based was 2.21 times the

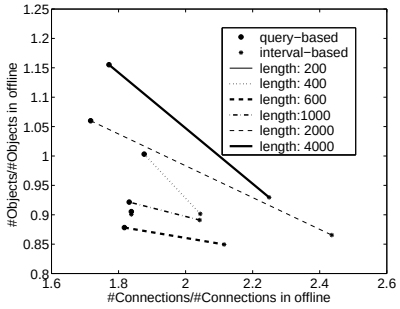


Figure 7: Number of connections Vs. Number of objects (as ratios to the offline) for varied degree of overlapping.

one of the offline. However, the number of transferred objects were 0.96 and 0.89 respectively.

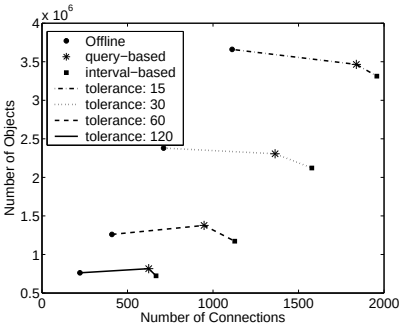


Figure 8: Number of connections Vs. Number of objects for tolerance varied from 15 to 120 minutes.

To summarize, without prior knowledge of the values of α and β, it is difficult to determine which of the heuristics is better. This is due to the fact the query-based approach needs less contacts with the server but more objects to transfer. Thus, for a given α and β one should choose the preferred heuristic that minimizes the total cost.

6.3 Heuristics Vs. Online Algorithm

Finally, we want to compare the heuristics with the online algorithm. The online algorithm depends on the values of α and β. Therefore, in the experiments, we show the total cost as a function of α and β. As a baseline for the comparison we will use the *Quasi-baseline* (Eq. 4). Figure 9 shows comparison of the communication cost to the quasi-baseline as a function of the ratio α/β. Using the same setting as in 6.2, with average queries length of 1,000 and average tolerance of 24 minutes. We let the ratio varied from 50 to 2000. For $\alpha/\beta = 1000$, the total communication cost of the online was 2.5 times the quasi-cost, for query-based and interval-based the cost was 2.3 and 2.2 times the quasi-cost respectively. We can clearly see that neither algorithm is dominant. The online algorithm performs the best for $\alpha/\beta < 200$ (***GreedyB***) and then again when $\alpha/\beta > 1800$ (***GreedyA***), then for $200 < \alpha/\beta < 1600$ interval-based was the best, finally for $1600 < \alpha/\beta < 1800$ the query-based dominates the others.

The algorithms' non-dominance serves as a motivation for designing an adaptive heuristic that can detect the best heuristic in a given setting and is one of our tasks for future work.

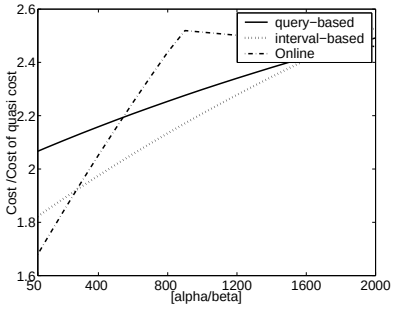


Figure 9: Cost (as ratio of the quasi-cost) Vs. α/β.

7. CONCLUSIONS AND FUTURE WORK

In this paper we study different techniques for reducing communication cost by pre-fetching data in a client-server environment. We gave algorithms with bounds followed by heuristics that were shown empirically to be useful.

This paper opens many new research directions. Currently we are working on a formal model for the query sequences, the identification of model parameter values for which different heuristics perform better, and extending the model for different quality measures such as decay cost, etc.

Acknowledgment

We thank Haggai Roitman for useful comments.

8. REFERENCES

[1] L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the web, 2002.

[2] D. Carney, S. Lee, and S. B. Zdonik. Scalable application-aware data freshening. In *ICDE*, pages 481–492, 2003.

[3] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. *SIGMOD*, 2000.

[4] E. Cohen and H. Kaplan. The age penalty and its effect on cache performance. pages 73–84.

[5] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[6] J. Eckstein, A. Gal, and S. Reiner. Optimal information monitoring under a politeness constraint. *INFORMS Journal on Computing*, 2007.

[7] A. Gal and J. Eckstein. Managing periodically updated data in relational databases: a stochastic modeling approach. *Journal of the ACM*, 48(6):1141–1183, 2001.

[8] M. J. Katz, J. S. B. Mitchell, and Y. Nir. Orthogonal segment stabbing. *Comput. Geom. Theory Appl.*, 30(2):197–205, 2005.

[9] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(1):035–047, 1996.

[10] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries. In *PODS*, pages 196–204, 2000.

[11] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 73–84, 2002.

[12] Y. Tao and D. Papadias. Adaptive index structures. In *VLDB*, pages 418–429, 2002.

[13] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.