

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

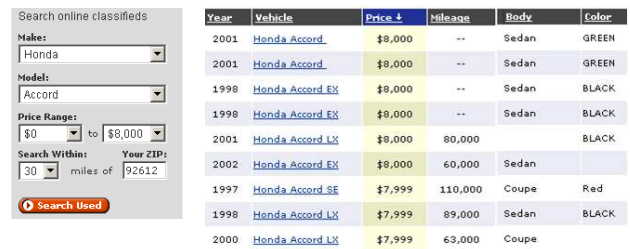
ABSTRACT

We study how to reduce costs in client-server 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. Further, allowing some tolerance towards data staleness in answering 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 with a lower cost. In this paper we develop algorithms under different cost models. In particular, for a generalized cost model, we provide a 2-approximation offline algorithm, a competitive online algorithm, and a family of heuristics. We validate our methods through extensive experiments.

1. INTRODUCTION

We consider client-server applications in which 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, 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 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



Year	Vehicle	Price ↓	Mileage	Body	Color
2001	Honda Accord	\$8,000	--	Sedan	GREEN
2001	Honda Accord	\$8,000	--	Sedan	GREEN
1998	Honda Accord EX	\$8,000	--	Sedan	BLACK
1998	Honda Accord EX	\$8,000	--	Sedan	BLACK
2001	Honda Accord LX	\$8,000	80,000		BLACK
2002	Honda Accord EX	\$8,000	60,000	Sedan	
1997	Honda Accord SE	\$7,999	110,000	Coupe	Red
1998	Honda Accord LX	\$7,999	89,000	Sedan	BLACK
2000	Honda Accord LX	\$7,999	63,000	Coupe	

(a) A query using search form

(b) Query results

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

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.

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 (such as age of data as formally defined in Section 3.1), then the client can just ask the server the corresponding cars in the range of \$8000 and \$9000. Such a decision depends on the freshness requirement on the answer to the query.

In this paper we study research challenges related to reducing costs in client-server applications with the following four characteristics, as illustrated by the example above.

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 pre-defined 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 on a query basis, as opposed to communicating on an object basis, i.e., queries

for a specific object. The client issues a query with semantic conditions, and the server returns objects satisfying these conditions.

In this work we make the following contributions.

- We present the problem setting, including issues such as how to define quality of query answers and different communication cost models (Section 3).
- We develop offline and online algorithms for reducing communication costs to answer queries to meet their freshness requirements under different cost models (Section 4). We present a 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 missing data. 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 [9], the server is assumed to be 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] study, given a query workload, how to decide on a synchronization policy to maximize some freshness measure under a limited bandwidth. [1] considers 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 applications, the server is 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. In addition, a model of object updates (such as the one in [7]) can help us provide such a guarantee.

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 of this answer based on the quality of these objects. Formally, let S be a set of objects (at the client side) satisfying the query conditions. For each object in S , we maintain its age. We define $age(Q, S)$ (or simply $age(Q)$ when S is clear from the context) as the age of the oldest object in S . The quality requirement of Q is satisfied if $age(Q) \leq \tau(Q)$.

Figure 2 shows an example workload with 4 range queries and their arrival times, each with a tolerance of $\tau(Q_i) = 7$ time units. For example, query $q2$ arrives at time $t = 5$ and asks for objects with a price ranging from $10K$ to $40K$. It allows a tolerance of 7 time units. The numbers of objects in the price ranges $[10K, 20K]$, $[20K, 40K]$, and $[40K, 50K]$ are 4, 6, and 5, respectively.

3.2 Cost Measures

We consider the cost (to answer a query) of the form

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

where α is a coefficient that captures *query-independent* costs

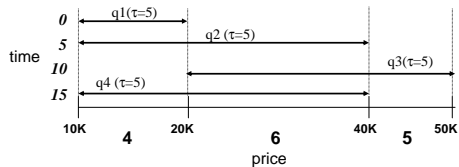


Figure 2: Range queries with their arrival times and quality requirements.

such as latency, β is a coefficient that captures *query-dependent* costs such as network bandwidth, and N is the size of the transferred data to answer the query. The cost of a set of queries is the summation of the costs of these queries.

In this work we 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, i.e., $\alpha \neq 0$ and $\beta = 0$. This model is realistic 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. The second simpler model is for applications where the size of the transmitted data is important. We set $\alpha = 0$ and $\beta \neq 0$, and our goal is to minimize the size of transmitted data.

3.3 Optimization Problem

We study the following problem. Consider queries with conditions on a numeric attribute. Each query has a *range condition* for this attribute, and asks for information about objects satisfying the range condition. A query also specifies its *quality requirement*. For a given query workload, we want to decide the times and the corresponding sub-queries the client needs to send to the server in order to minimize the total communication cost, subject to the quality requirement.

4. ALGORITHMS

In this section we develop algorithms for the optimization problem in Section 3.3 under various cost models presented in Section 3.2. 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 result is significant since we show that every online algorithm has a competitive ratio that is at least half the cost ratio.

4.1 Simple Cost Models

Model 1: $\alpha \neq 0$ and $\beta = 0$: In this cost model, consider a greedy online algorithm, denoted *GreedyA*, which contacts the server whenever the quality requirement for a query cannot be satisfied using the local data at the client, and asks for all the objects. It is easy to show that this algorithm generates an optimal plan. The complexity of the algorithm is $O(n \log n)$, where n is the number of queries [8]. Figure 3(a) shows the times and the sub-queries for the example from Figure 2. In the example, we assume that each interaction with the server costs 10, i.e., $\alpha = 10$. For ease of exposition, we assume that between different interactions, 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, independently of the size of the downloaded data.) Therefore, the total cost for this plan is 20.

Model 2: $\alpha = 0$ and $\beta \neq 0$: In this cost model, consider a greedy online algorithm, called *GreedyB*, in which the client contacts the server whenever its local data cannot satisfy the quality requirement of a query, and asks for the needed ranges only. It can be shown that this greedy algorithm yields an optimal solution. The complexity of the algorithm is $O(n \log n)$, where n is the number of queries [8]. For example, Figure 3(b) illustrates the times and the corresponding sub-queries. Assuming the cost of transmission per object is 2, i.e., $\beta = 2$, and we only have price-change updates, the total cost of the shown plan is 50, which includes cost 4 at time $t = 0$, cost 6 at $t = 5$, cost 5 at $t = 10$, and cost 10 at $t = 15$.

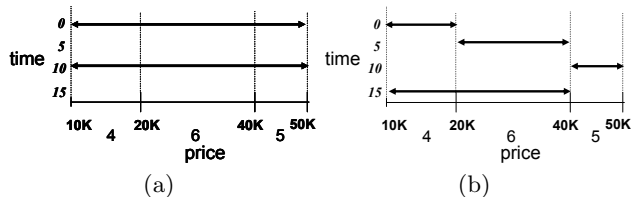


Figure 3: Running greedy algorithms on the example workload. (a) *GreedyA*-optimal plan, total cost of 20; (b) *GreedyB*-optimal plan, total cost of 50.

4.2 Generalized Cost Model: Offline Algorithm

We show how to generalize these two algorithms above into one that considers both the constant overhead (α) and the data-dependent cost ($\beta \times N$). We first consider the offline case in which we assume that the client has full knowledge about the queries, including their arrival times, condition ranges, and quality requirements. Recall that the output of our 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. It combines algorithms *GreedyA* and *GreedyB* in the following manner. The contact time instances are derived from algorithm *GreedyA*, and the sub-queries are derived from *GreedyB*. Sub-query on which both greedy algorithms agree (i.e., fired at the same time instance), we add the sub-query to the final plan. Sub-queries (of *GreedyB*) in which the two greedy algorithms disagree, it will replace the sub-queries with two queries, one will be added to the “previous” contact time instance and another to the “next” contact time. If any such query has already been chosen in a previous step, that query is skipped.

Figure 4 shows how this algorithm constructs the offline plan. For example, the two greedy algorithms agree on (time $t = 0$, sub-query $[10K, 20K]$). Thus we add this sub-query to the plan (depicted by the solid edge). However, they disagree for (time $t = 5$, sub-query $[20K, 40K]$). Thus we duplicate the sub-query, adding it to the previous contact time $t = 0$, and to the next contact time $t = 10$ (depicted by the two dashed edges). For (time $t = 15$, sub-query $[20K, 40K]$), we should duplicate the sub-query and add it to time $t = 10$ and for a future time, which does not appear in the figure. Since

we have already added this sub-query to the plan at $t = 10$ (because of previous queries), 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.

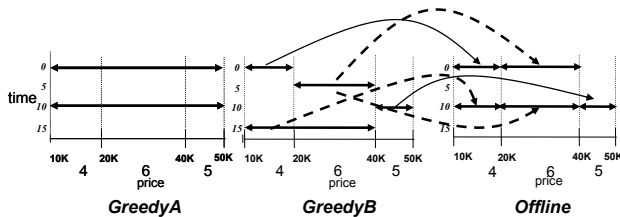


Figure 4: Construct a 2-approximate plan with a total cost=70. An optimal solution also has a cost of 70.

THEOREM 4.1. *The offline algorithm described above is 2-approximate under the general cost model.*

4.3 Generalized Cost Model: Online Algorithm

We develop 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 as:

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

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 , we can show that the maximum possible value of the cost ratio is \sqrt{L} (when $\alpha/\beta = \sqrt{L}$).

Algorithm Ratio(α, β, L)

If($\alpha/\beta \geq L\beta/\alpha$)

Use algorithm GreedyA.

Else

Use algorithm GreedyB.

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

The algorithm **Ratio** is described in Figure 5. It simply chooses between GreedyA and GreedyB based on the cost ratio. We can bound the competitive ratio of **Ratio** as stated below.

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

This algorithm is 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 of at least $(k+1)/2$, where k is the cost ratio.*

5. HEURISTIC-BASED APPROACHES

The offline algorithm assumes knowledge of the entire sequence in advance, and this assumption might not be true in some applications. Online algorithms do not have such requirements. They, however, are designed to optimize for worst-case sequences, which occur infrequently in real-world applications, and so are considered pessimistic. Any algorithm that assumes that the sequence will not turn out to be one of the worst-case ones, can make “risky” decisions that are better than the online algorithm 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.

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 time window, we maintain an average tolerance of the query instances, denoted $AVG(Q)$. For example, consider a query Q : “find cars with a price in $[16K, 18K]$.” Suppose it has two instances, Q' and Q'' , with the corresponding tolerances $\tau(Q') = 10$ minutes and $\tau(Q'') = 20$ minutes. 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 at the client in order to answer the query (as defined in 3.1). The pre-fetch rule is:

$$\text{If } age(Q) > AVG(Q) \text{ then } prefetch(Q). \quad (2)$$

5.2 Interval-based Heuristic

Using this heuristic, we divide the attribute domain (for example, the car’s price) 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:

$$AVG(i) = \frac{\sum_{Q \text{ overlap interval } i} Q(\tau)}{\sum_{Q \text{ overlap interval } i} 1}$$

The pre-fetch rule is

$$\text{If } age(i) > AVG(i) \text{ then } prefetch(i), \quad (3)$$

where $age(i)$ is the age of the oldest object in interval i . For example, a query Q is “find cars priced between $[16K, 18K]$,” and its tolerance $\tau(Q) = 10$ minutes. There is another query Q' : “find car priced between $[17K, 20K]$,” and its tolerance is $\tau(Q') = 20$ minutes. Assuming these are the only queries that overlap interval $[17K, 18K]$, then the average tolerance is $AVG(i) = 15$ minutes. If the age of the oldest object within this interval is greater than 15 minutes, then we prefetch the data for this interval $[17K, 18K]$.

6. EXPERIMENTAL RESULTS

In this section, we present experimental results that demonstrate the usefulness of data caching and pre-fetching in reducing communication costs, and compare the performance of the various approaches we proposed in the earlier sections for different scenarios. We used synthetic workloads generated with controlled variations in the number of queries, the number of objects, and query lengths. 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 were generated using an exponential distribution. For each setting, we ran the experiments on 100 different workloads and computed the average. The results were very stable.

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

6.1 Offline Algorithm versus Online Algorithm

The offline algorithm is 2-approximation, so in the worst case its cost is twice the optimal algorithm. The online algorithm Ratio is $(k + 1)$ -competitive, so in its worst case the cost will be $(k + 1)$ times the optimal. Thus, it is possible that for different sequences the offline algorithm will perform better than the online algorithm 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 algorithm outperforms the online *GreedyA* algorithm, while Figure 6(b) shows the opposite case in which the online *GreedyB* algorithm is better than the offline algorithm.

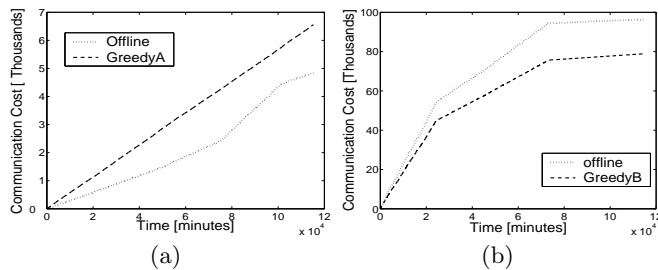


Figure 6: 10000 queries, 80 days. (a) $\alpha = 1, \beta = 0.01, L = 100$; (b) $\alpha = 1, \beta = 0.2, L = 200$.

In order to evaluate the performance of the heuristics versus the online algorithm, we must overcome this “flip-flop phenomena” seen above. We define a fixed baseline, denoted *Quasi-baseline*. For each time t , the baseline cost is as follows:

$$cost_quasi(t) = \max \left\{ \frac{cost_online(t)}{k + 1}, \frac{cost_offline(t)}{2} \right\}. \quad (4)$$

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 versus Offline Algorithm

The heuristics are independent of specific values of α and β . Thus, we can evaluate their behaviors with respect to other parameters such as query length and tolerance. In the first set of experiments, we varied the query 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 be $W = 2$ days. We let the query length vary from 200 to 4,000. Table 1 shows the number of connections and the amount of transferred data for the heuristics. For a small degree of overlapping (e.g., length=200), each query covered a unique interval, thus the heuristics converged to a similar solution. However, when the query length increased such that each query overlapped with more intervals, the query-based heuristic consistently contacted the server fewer times than the interval-based heuristic, but each time transferred more objects. This is due to the fact that the interval-based heuristic corresponds to sending the remainder queries while the query-based heuristic sends whole queries.

query length	query-based		interval-based	
	Connections	Objects	Connections	Objects
200	1,522	579,800	1,524	576,702
400	1,548	1,028,792	1,686	924,578
600	1,503	1,694,400	1,749	1,638,864
1000	1,524	2,770,169	1,699	2,678,217
2000	1,406	4,262,425	1,995	3,480,080
4000	1,458	7,119,728	1,851	5,728,601

Table 1: Number of connections and transferred objects for query and interval based heuristics.

To evaluate the goodness of the heuristics, we compared 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 the query-based heuristic and the interval-based heuristic was 1.83 times and 2.04 times that of the offline algorithm, respectively. The number of transferred objects of these two heuristics was 0.92 and 0.89 times of the offline algorithm, respectively.

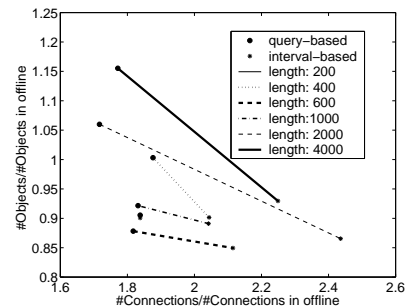


Figure 7: Number of connections versus number of objects (as ratios to the offline algorithm) for different degrees of overlapping.

To verify the results, we conducted another set of experiments in which we varied the tolerance of 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 query length 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 the cases, we can see that the results are consistent with our observation on the behavior of the interval and query-based heuristics: the query-based heuristic always has fewer connections and more transferred data compared to the interval-based heuristic. The figure also shows the number of connections and number of transferred objects of the heuristics compared to the offline algorithm. For instance, when the tolerance was 30, the number of connections for the query-based heuristic and the interval-based heuristic was 1.91 times and 2.21 times that of the offline algorithm, respectively. The number of transferred objects of these two heuristics was 0.96 and 0.89 times that of the offline algorithm, respectively.

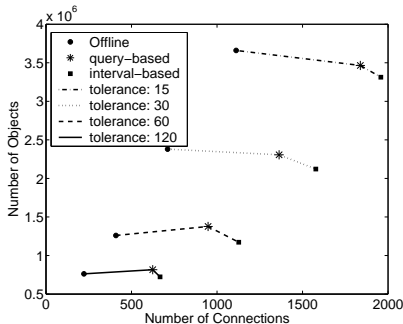


Figure 8: Number of connections versus number of objects for tolerance varied from 15 to 120 minutes.

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 heuristic needs fewer 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 versus Online Algorithm

Finally, we compared 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 use the *Quasi-baseline* (Eq. 4). Figure 9 shows the comparison of the communication cost to the quasi-baseline as a function of the ratio α/β . Using the same setting as in Section 6.2, with average query length of 1,000 and average tolerance of 24 minutes. We let the ratio vary from 50 to 2000. For $\alpha/\beta = 1000$, the total communication cost of the online algorithm was 2.5 times the quasi-cost, for query-based and interval-based heuristics the cost was 2.3 and 2.2 times the quasi-cost, respectively. We 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*). For $200 < \alpha/\beta < 1600$, the interval-based heuristic was the best. Finally for $1600 < \alpha/\beta < 1800$, the query-based heuristic 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.

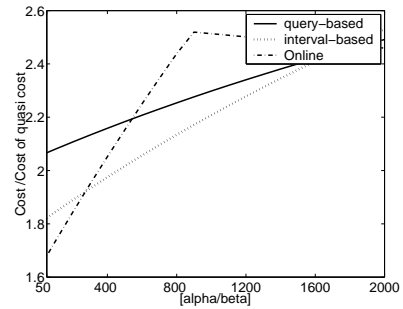


Figure 9: Cost (as ratio of quasi-cost) versus α/β .

7. CONCLUSIONS

In this paper we developed different techniques for reducing communication costs in client-server applications by prefetching data to answer queries. We proposed algorithms with provable bounds followed by heuristics that were shown empirically to be useful.

Acknowledgments

We thank Haggai Roitman for providing 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*, 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.