Achieving Communication Efficiency through Push-Pull Partitioning of Semantic Spaces in Client-Server Architectures

(UCI ICS Technical Report, February 2005)

Amitabha Bagchi
Polytechnic University, Brooklyn, NY 11201, USA

Amitabh Chaudhary
University of Notre Dame, IN 46556, USA

Michael T. Goodrich, Chen Li, and Michal Shmueli-Scheuer
UC Irvine, CA 92697, USA

Abstract

Client-server databases that require query results to be up-to-date despite storing data that changes dynamically suffer from heavy communication costs. Client-side caching can help mitigate these costs, particularly when individual PUSH-PULL decisions are made for the different semantic regions in the data space. In the PUSH regions the server notifies the client about updates, and in the PULL regions the client sends queries to the server. We call the problem of partitioning the data space into PUSH-PULL regions to achieve the minimum possible communication cost for a given workload the problem of *data gerrymandering*. In this paper we present solutions under different communication cost models for a frequently encountered scenario: with range queries and point updates. Specifically, we give a provably optimal-cost dynamic programming algorithm for gerrymandering on a single range query attribute. We propose a family of heuristics for gerrymandering on multiple range query attributes. We also handle the dynamic case in which the workload evolves over time. We validate our methods through extensive experiments on real and synthetic data sets.

1 Introduction

The more databases give us, the more we demand from them. In the context of client-server based applications, we want up-to-date query results even in the face of rapidly changing data. Client-side caching has emerged as a key technique for reducing the heavy communication costs that result, making faster and better what was once impossible. A typical example is Carfaire.com, an online car-buying-and-selling service sponsored by a group of community colleges in California that uses a client-server architecture. At Carfaire, sellers can add their vehicles to the database and delete them once they are sold. Buyers, meanwhile, can search for cars satisfying conditions such as "manufactured between 1990 and 1995, and priced between \$5,000 and \$10,000." Communication costs can be in terms of initial delay or limited bandwidth. Having low communication costs, is vital; Carfaire is a crowded site during weekends, and no buyer is willing to miss out on a hot deal just because the system cannot handle the traffic. Client side caching [KB96, Fra96] can be used to alleviate the costs.

In this paper we focus on finding the *minimum possible communication cost* that can be achieved using client-side caching in a client-server application. Let us begin by understanding intuitively how client-side caching works. The client caches a local copy of part of the data that is queried frequently. As a result, some queries can be answered using the data in the cache without any communication with the server. Keeping the cached data up-to-date is, however, not a simple task and can lead to considerable communication cost. Consider the case in which all the queries originate at the client and all the updates at the server. If the queries at the client are "far more" than the updates at the server, then to keep the cached data updated, the server can just send all the updates it receives to the client. The client incorporates updates into its cached copy, and uses it to answer queries. This is called the PUSH communication paradigm. However, when the queries at the client are much "fewer" than the updates at the server, this paradigm is inefficient. Instead, we prefer the PULL communication paradigm: the server applies the updates to its own copy of the data, and the client sends each of its received queries to the server. The server processes the query and sends back the results. It is not difficult to see that any mechanism that manages updates and queries in a client-server application using client-side caching employs a PUSH or a PULL philosophy or a combination. (PUSH-PULL techniques have been well studied in the context of broadcast disks, continuous media streaming, web-based caching, etc. See, e.g., [Fra96, OLW01, DKP$^+$01, SRB97].) In general, the relative number of queries and updates varies along the data space. Different regions of the data space may need different PUSH-PULL choices, and making the best set of choices becomes an optimization problem.

This leads to the main question tackled in this paper: Given a workload of queries and updates in the data space, how can we best partition the space and make PUSH-PULL decisions for each partition so as to minimize communication cost? We call this problem *data gerrymandering*. The name is inspired by the concept of gerrymandering electoral districts to consolidate support for one political party. Here we want to gerrymander data, i.e., partition it to combine regions that make the same PUSH-PULL choice.

It is worth mentioning that optimal partitions of the data space are those made along *semantic* boundaries rather than along page or object-ID boundaries. Intuitively, semantic boundaries are those created by the queries on the data space, i.e., tuples that remain together in query results are in the same semantic region. In Section 3.2 we see how the semantic regions for the queries we are interested in get defined.

In this paper, we focus on data gerrymandering for a workload consisting of range queries and point updates. There are several important applications other than car trading sites that have such workloads, e.g., traffic information systems, such as the Travel Advisory News Network (TANN)[1], which provide real-time traffic information. Using their service, a mobile driver can ask questions such as "Are there any accidents between Exit 3 and Exit 30 on highway 95?" It is obvious why the results need to include the latest updates. Other example applications that fall in our framework include astronomy databases such as the Sloan Digital Sky Survey [sdsa] and the World Wide Telescope [GS01]; and flight and hotel booking systems such as Orbitz.com.

We assume that cache size is not an issue: the client can store as large a data set as necessary. This is increasingly true with lowering storage costs. For instance, in many mediation systems [Wie92], the mediator has enough space to store the entire data from a source. Even with this assumption, data gerrymandering remains difficult and its solutions are non-trivial. We focus on the single client, single server problem, with all the range queries originating at the client and all the point updates at the server. If there are multiple clients accessing the server, each client solves, independently, an instance of the data gerrymandering problem based on the workload it encounters.

A solution to the data gerrymandering problem returns a partition of the data space and a PUSH or PULL label for each partition, based on the given workload. With such a solution the client and server follow a simple protocol for maintaining an updated cache and for answering queries. All data in the PUSH regions is copied to the client cache. Data in the PULL regions is stored at only the server. On receiving a point update the server incorporates it into the server data, and if the update falls inside a PUSH region, it sends the update to the client, who incorporates it into its cache. On receiving a range query, if the query does not intersect any PULL region, the client answers the query using just the cached data. Otherwise, the client sends the query to the server, which processes it using server data and sends the results back to the client.

Our contributions towards solving the problem of data gerrymandering follow:

- We motivate data gerrymandering through an example, showing how finding the best solution is non-trivial, yet useful (Section 2).

[1]http://traffic.tann.net/

- We study the optimization problem of finding an optimal partition with PUSH/PULL labeled regions on a single gerrymandering attribute. We develop a dynamic programming algorithm for finding an optimal labeled partition under a simple communication cost model, and extend it to general cost models (Section 3).
- We show how to *efficiently* gerrymander both in the single-attribute case and in the multiple-attribute case. We develop a family of heuristics for finding a good labeled partition, using which we can effectively choose gerrymandering attributes (Section 4).
- We study how to do adaptive gerrymandering as the distribution of queries and updates changes (Section 5).
- We conduct extensive experiments on both synthetic data and real data to evaluate different gerrymandering algorithms (Section 6).

1.1 Related work

Semantic data caching was proposed in [DFJ$^+$96], in which this the client maintains a semantic description of the data in its cache, and decides whether it should contact the server for data not in the cache. The data needed from the server is specified as a reminder query. [DFJ$^+$96] focused on cache replacement policies, without considering data updates on the server. We consider the case where the client site has enough cache to store data, and the server data is dynamic.

Our work is closely related to the predicate-based caching scheme proposed in [KB96], which uses possibly overlapping query-based predicates to describe the cached data on the client. The server is responsible for notifying the client about data updates satisfying these predicates. There are two main differences between our work and theirs. Firstly, our work uses the concept of a PUSH/PULL partitioning scheme for the purpose of formalizing the notion of using semantic regions to minimize communication costs. In our work, finding a good partitioning scheme requires a sophisticated communication model based on the distribution of queries and updates. Developing this allows us to systematically study the problem of minimizing communication cost. Second, the work in [KB96] does not include any experimental evaluation to validate their heuristics. We develop efficient algorithms to solve the gerrymandering problem, including algorithms adopted from the heuristics proposed in [KB96] (Section 4.1). We conduct an intensive experimental study on different algorithms.

Among other related work is web caching [Wan99, AWY99, Fra96]. One line of work proposes that each document at the client should have its own associated strategy for deciding whether to be pushed to a replication site or stay at the server [PvST02]. Other techniques take the capabilities and load at the servers and proxies, and clients' coherency requirements to adaptively decide whether data should be pushed or stored till it is pulled [DKP$^+$01]. Some use data affinity to reduce cache misses [Ji02].

The application of the notion of semantic regions to caching seems to be a natural notion which has been studied in other contexts. One example is Amiri et al.'s proposal for Content Distribution Networks [APTP03]. They suggest a three-level architecture — webserver, application server, database — and propose a method for storing useful data at the application server to speed up web applications. In the context of query processing, Haas et al. [HKU99] consider the situation where queries may need to go back to the database to execute methods on objects they have accessed already. In this situation they suggest that the client improves performance significantly by fetching and storing data that it might need to completely process the query.

There has also been interest in making broadcast environments adaptive by allowing clients to request data back channel [AFZ97, KCJH01]. The asymmetry inherent in the capabilities of nodes versus base stations in mobile networks makes this kind of capability particularly important [Bar99]. Our approach involves partitioning the space into regions and studying their properties to make intelligent dissemination. One area of research which proceeds on similar lines involves constructing histograms over the range based on different parameters. For example, [BCG01] described a method of forming workload-aware histograms based on the density of queries over the range. Another related area is to answer client queries using cached data with certain errors [OLW01]. The difference is that our work assumes each query should be answered using the latest data without any error. A similar idea of combining PUSH and PULL paradigms is developed in [TYD$^+$04] to reduce communication cost to answer queries in sensor networks.

2 Problem Motivation and Formulation

In this section we use an example to give an overview of data gerrymandering.

2.1 Motivating Example

As an illustrative example, consider an online car shop that integrates car information from different dealer sources. The shop receives user queries and contacts the sources to retrieve related data. The shop can be viewed as a client, and each source is a server that provides data and answers queries sent from the client. Data gerrymandering can be used between the shop (client) and each source (server). Let us consider one particular source, whose car information is stored in a relational table with the schema `car(make, model, year, mileage, color, price)`. The following are example client queries.

- Find cars satisfying `year` $>$ `1998 & price in (5K, 9K)`;
- Find the average price of cars satisfying: `color = 'red' & mileage in (30K, 50K)`;
- Find the number of cars satisfying: `model = 'Camry' & mileage` $<$ `90K & price in (3K, 8K)`.

We use the `price` attribute to illustrate the advantages of data gerrymandering. Fig. 1 shows a workload of the data updates at the source and the queries sent to the source during a certain time period. For instance, there are 2 car updates (u_1 and u_2) in the price range $(2, 6)$ (all in thousands), e.g., new cars are inserted or existing cars are deleted. Similarly, there is 1 car update (u_3) in range $(6, 10)$, 2 car updates (u_4 and u_5) in range $(10, 14)$, and 2 updates (u_6 and u_7) in range $(14, 18)$. In addition, during this period users posed 6 queries $q_1, \ldots, q_6$. For instance, query q_1 asked for cars in the price range $(6, 18)$, and query q_2 asked for cars in the range $(2, 14)$.

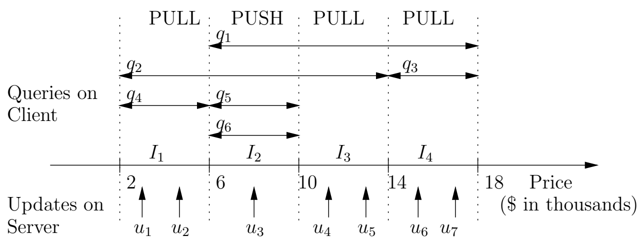


Figure 1: Queries and updates on cars by price.

We want to answer these queries using the latest data at the server (the source) while reducing the communication cost. If we use the PUSH method for the entire price domain, the server needs to send each update to the client as soon as the update occurred. As a consequence, the server needs 7 interactions with the client for the 7 updates. On the other hand, if we use the PULL method, the client needs to contact the server for each user query. Thus the client needs to have 6 interactions with the server for the 6 queries. If we want to minimize the number of interactions between sides (without considering the size of the transferred data), it is better to use the PULL method due to its lower cost.

Interestingly, we can further reduce the communication cost by carefully choosing PUSH and PULL methods for different intervals. In particular, suppose we use the PUSH method for interval $I_2 = (6, 10)$ and the PULL method for the other three intervals I_1, I_3, and I_4. In addition, the client and the server follow a simple protocol. On the server site, any update in a PUSH interval (u_3 in this example) needs to be sent to the client, which caches all the latest data in the PUSH intervals. An update in a PULL interval (e.g., u_1 and u_2) does not need to be propagated to the client. On the client site, any query overlapping with a PULL interval (q_1, q_2, q_3, and q_4) needs to be answered by contacting the server to the get latest data; any query not overlapping with any PULL interval (q_5 and q_6) can be answered using the data on the client without contacting the server. In summary:

Server, pushed updates: u_3.
Server, ignored updates: u_1, u_2, u_4, u_5, u_6.
Client, queries contacting Server: q_1, q_2, q_3, q_4.
Client, queries answered locally: q_5, q_6.

Using this PUSH/PULL labeling, the total number of interactions between the two sides is reduced to 5. Notice that this reduction can be arbitrarily large if we increase the number of queries and updates in the workload. This simple example shows that by carefully partitioning the attribute domain and choosing a PUSH/PULL mode for each interval, we can reduce the total communication cost. This idea is called data gerrymandering.

2.2 Data Gerrymandering

We now formally describe data gerrymandering ("gerrymandering" for short). Consider a client-server environment, where the server maintains a collection of objects (e.g., information about cars), and the client receives queries to be answered using the latest data on the server. In the case of multiple clients and servers, each client-server combination can adopt the gerrymandering technique. The data on the server is dynamic — new objects can be inserted, and existing objects can be deleted or modified. Each update is a *point update*, which affects only one object. Each query specifies conditions on these attributes, and asks for information about objects satisfying these conditions, such as all these objects or an aggregated value (SUM, AVG, COUNT, MIN, MAX, etc).

Data gerrymandering: the client and the server decide a collection of semantic regions, each of which has a PUSH or PULL label. The client caches the data in the PUSH regions, and the server notifies any data change in these PUSH regions. Client queries overlapping PULL regions need to be answered by contacting the server.

Specifically, the client and the server first choose a subset of the data attributes. The domain of each attribute is divided into non-overlapping regions. For a numeric attribute, we can partition the domain into intervals, as we did for the car price attribute in the example above. The gerrymandering technique is applicable to various types of attributes, as long as the client and the server can agree on a partition on the domain of an attribute. For instance, for a categorical attribute such as car make, a region could be a single value (e.g., "BMW," "Ford," "Honda," and "Kia"), or a collection of values such as American car makes, Japanese car makes. For simplicity, in this paper we make the following assumptions:

- We gerrymander numeric attributes only.
- Each query condition on a numeric attribute is a range in the format of (a, b), $(a, b]$, $[a, b)$, or $[a, b]$. If $a = b$, we call it a *point range* or a *point interval*.

Semantic regions are decided *a priori* by the client and server together by partitioning each attributes range into intervals and taking the Cartesian product of these intervals. In addition, both sides decide a PUSH or PULL label for each semantic region. In the example above, we chose only one attribute, `price`, as the gerrymandering attribute, and decided a PUSH/PULL label for each interval. In general, multiple attributes can be used in data gerrymandering, such as $\{$price, year$\}$ or $\{$price, year, mileage$\}$. Fig. 2 shows a labeled partition for attributes $\{$price, year$\}$. Example semantic regions are "price in (2K, 6K] & year in (1999, 2001]" (labeled as PUSH) and "price in (6K, 8K] & year in (1994, 1997]" (labeled as PULL).

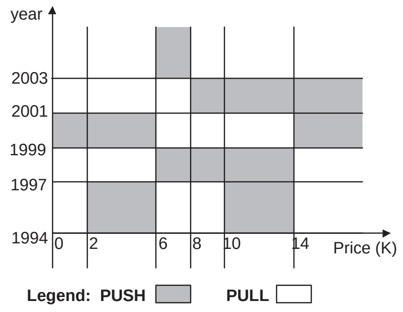


Figure 2: A labeled partition for attributes $\{$price, year$\}$.

3 Gerrymandering on a Single Attribute

In this section we study the following problem: given a workload of queries and updates, how do we find an optimal labeled partition of a single attribute with minimum communication cost? We formulate an optimization problem, and develop a dynamic programming algorithm for finding an optimal solution. The algorithm is developed under a simple cost model, and extended to two more general cost models.

3.1 Optimization Problem

Consider a numeric attribute which we want to gerrymander. The attribute has a totally ordered domain between a lower bound (possibly $-\infty$) and an upper bound (possibly $+\infty$). Each query has a *range condition* for this attribute, and asks for information about objects satisfying the range condition. We are given a workload, which is a set of queries and updates. We want to partition the domain of this attribute into a set of disjoint intervals and assign each interval a PUSH/PULL label in order to minimize the communication cost for this workload.

We are given a *communication cost model* $\mathcal{M}$ that does the following. Let P be a labeled partition for a workload W. For each update u_i (in W) in a PUSH interval, the model $\mathcal{M}$ returns the communication cost (denoted $c(u_i)$) when the server sends this update to the client immediately after u_i happens. Using gerrymandering, the communication cost of an update in a PULL interval is 0. For each query q_j (in W) intersecting a PULL interval, the model $\mathcal{M}$ returns the communication cost (denoted $c(q_j)$) when the client contacts the server to get necessary data to answer q_j. The communication cost of a query not intersecting any PULL interval is 0. Notice a query q_j intersecting a PULL interval may also overlap with some PUSH intervals. Since the client has up-to-date data for these PUSH intervals, the sever does not need to send data in these intervals.

If we denote the set of updates falling in intervals labeled PULL under P as U_P and the set of queries overlapping intervals labeled PULL as Q_P, then we have that the overall cost of workload W under labeling P is given by:

$$Cost_P(W) = Cost_U(U_P) + Cost_Q(Q_P)$$

The three cost models we consider in this paper have the additional property that the costs of individual updates and queries add up linearly. In other words, if for an update u we denote $Cost_U(\{u\})$ by $c(u)$ and for a query q we denote $Cost_Q(\{q\})$ by $c(q)$), we have:

$$\begin{aligned} Cost_U(U_P) &= \sum_{u_i \in U_P} c(u_i) \\ Cost_Q(Q_P) &= \sum_{q_i \in Q_P} c(q_j) \end{aligned}$$

This optimization can be reflected in the cost model.

Optimization Problem: Given a workload W of queries and updates and a communication cost model $\mathcal{M}$, find a labeled partition P for the gerrymandering attribute to minimize the communication cost $Cost(P)$.

Notice that in this setting, *the labeled partition between the client and the server does not change during the workload*. This implies that the order in which events happen in the workload do not affect the cost. In addition, note that the workload can incorporate not only cost-related information, but also the probability of occurrence of each event, if available.

3.2 Deciding Intervals of Interest

We first identify the intervals that need to be considered in order to find an optimal labeled partition.

Proposition 3.1 *(***Intervals of Interest***) Given a workload W with a set of updates U and a set of queries Q, let $p_1 < p_2 < \ldots < p_n$ be the starting and ending points of the ranges in the conditions of Q. (p_1 could be $-\infty$, and p_n could be $+\infty$.) Under any communication cost model, there is an optimal labeled partition P whose intervals are either of the form (p_i, p_{i+1}), or the form $[p_i, p_i]$.* □

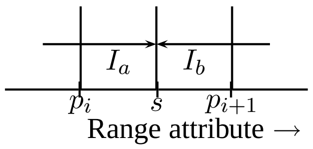


Figure 3: Intervals of interest.

Proof: As shown in Fig. 3, consider a labeled partition P and two ends p_i and p_{i+1} of the range conditions in the queries. Suppose P has two adjacent intervals I_a and I_b, whose common end s lies between p_i and p_{i+1}. Without loss of generality, assume I_a is in PUSH mode and I_b is in PULL mode. (If they have the same mode, we merge them into one interval with their mode, and repeat this process.) Now we modify P by merging I_a and I_b into one PULL interval and get a new labeled partition P'. Now we show under any cost model $\mathcal{M}$, we always have $Cost(P') \leq Cost(P)$. The reason is the following. (1) All the pulled queries in P still need to be pulled in P', since they still intersect a PULL interval in P'. (2) P' could even save the communication costs for the pushed updates (in P) in interval I_a, if any. We repeat this merging process until we get a new labeled partition that is in the format specified in the proposition. ■

Based on this proposition, we make the following simplifying assumptions about the workload for easy presentation. (1) No update occurs at the starting and ending points of the ranges in the conditions of the queries; and (2) each range condition is of the form (a, b). Based on Proposition 3.1, we only need to consider intervals of the form (p_i, p_{i+1}), and ignore point intervals.

A problem instance thus consists of a workload W with a set of range queries Q and a set of point updates U. Let the intervals of interests be $I_1, \ldots, I_n$ in ascending order. Each interval I_i is to be assigned to I_{PUSH} (a set of PUSH intervals) or I_{PULL} (a set of PULL intervals), so that the total communication cost is minimum. For convenience, we sometimes refer to I_j as interval j. We denote the number of updates in interval I_j by $U(j)$, and the number of queries intersecting interval I_j by $Q(j)$.

3.3 Model $\mathcal{M}_1$: Unit Cost

We first study the optimization problem for a simple cost model:

$$\boxed{\text{Cost model } \mathcal{M}_1\text{: Each communication between the client and the server has a cost of 1.}}$$

We consider this model because of its simplicity and applicability in many cases where the communication cost is mainly determined by the number of messages between the two sides. This cost model is relevant in high-bandwidth setting where latency is the main issue. We develop a dynamic-programming algorithm, called DYNPROG, for finding an optimal labeled partition.

We elaborate the basic idea using the car workload in Fig. 1. For interval I_1, we would have equal tendency to choose PUSH or PULL, since the number of updates $U(1) = 2$, and the number of intersecting queries $Q(1)$ is also 2. In particular, query q_4 lies entirely within the interval, and query q_2 overlaps with I_1, I_2, and I_3. If we choose a PULL mode for either I_2 or I_3, then the PULL cost for q_2 will be paid *irrespective* of the mode of I_1. On the other hand, in the case we mark I_2 as PULL, if we choose PUSH for I_1, we need to add 2 to the total cost for the 2 updates in I_1. If we mark I_1 PULL, we just add 1 to the total cost for q_4. In either case, the total cost includes the PULL cost for q_2. The same reasoning applies if we choose PUSH for I_2 and PULL for I_3, because q_2 intersects I_3 as well. Also notice that if we choose PULL for I_2, the decision for I_3 does not affect the decision for I_1. This example shows that the additional cost we need to pay for a query in an interval depends on the labels for other later intervals. In general, if a query in the current interval intersects other interval already marked PULL, its cost has already been "paid for" and should not be considered any more. Based on this observation, we use the power of dynamic programming to change our labeling decisions on the fly as we go through the intervals.

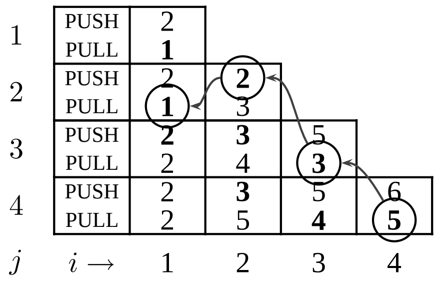
3.3.1 Subproblems and Recurrence Function

To capture the intuition above, we define the following optimization subproblems. For each interval i, for each interval j ($j \geq i$), we consider the following modified workload $W(i, j)$. It only includes the updates in intervals I_1 to I_i, and the queries that intersect an interval from I_1 to I_i, but do not intersect interval I_{j+1}. Workload $W(i, j)$ excludes all other updates and queries. Let $P(i, j)$ be the gerrymandering optimization subproblem for this modified workload, and $f(i, j)$ be the minimum cost for this subproblem. $f(i, j)$ can also be viewed as the minimum communication cost for the queries and updates in intervals I_1 to I_i, assuming that the next PULL interval after I_i is interval I_{j+1}. Clearly $f(n, n)$ is the minimum cost for the original optimization problem, where n is the number of intervals of interest.

DYNPROG is based on a recurrence for function $f(i, j)$. We consider two cases for the subproblem $P(i, j)$: interval I_i is in PUSH and I_i is in PULL.

$$\begin{aligned} i > 0 : f(i,j) &= \min \begin{cases} U(i) + f(i-1, j) & \text{PUSH} \\ Q(i,j) + f(i-1, i-1) & \text{PULL} \end{cases} \\ f(0, j) &= 0. \end{aligned}$$

In the recurrence, $U(i)$ is the number of updates in I_i, and $Q(i,j)$ is the number of queries intersecting interval I_i but do *not* intersect interval I_{j+1}. This is the number of queries left behind in the subproblem described above. Now we explain the recurrence function. Consider the case where I_i is in PUSH. This implies that the next closest PULL interval to I_{i-1} is I_{j+1}. Thus, the minimum cost for subproblem $P(i,j)$ in this case can be obtained by taking the summation of the minimum cost of the subproblem $P(i-1,j)$ (i.e., $f(i-1,j)$) and the cost of all updates in i (i.e., $U(i)$). Now consider the case where I_i is in PULL for subproblem $P(i,j)$. This implies that the next closest PULL interval to I_{i-1} is I_i. Since $Q(i,j)$ includes the costs for all queries intersecting interval I_i but not intersecting I_j, we can just consider the queries only intersecting intervals from I_1 to I_{i-1}. In this case, subproblem $P(i,j)$ can be solved by solving the subproblem $P(i-1,i-1)$, and then adding the cost of $Q(i,j)$. Finally, $f(i,j)$ should be the minimum of the costs in these two cases.



j	$i \rightarrow$	1	2	3	4
1	PUSH	2			
	PULL	**1**			
2	PUSH	2	**2**		
	PULL	**1**	3		
3	PUSH	**2**	**3**	5	
	PULL	2	4	**3**	
4	PUSH	2	**3**	5	6
	PULL	2	5	**4**	**5**

Figure 4: Computing an optimal solution for the example in Fig. 1.

We use Fig. 4 to show how to compute the $f(i,j)$ values for the gerrymandering problem in Fig. 1. Each entry (i,j) in the table has two values, where the "PUSH" value (resp. the "PULL" value) corresponds to the minimum cost if interval I_i is in PUSH (resp. PULL) for the subproblem $P(i,j)$. The minimum value (in bold) of the two is the $f(i,j)$ value. For easy computation, we assume $f(0,j) = 0$. For instance, consider the entry $(1,1)$. The PUSH value is $U(1) + f(0,1) = 2 + 0 = 2$. The PULL value is $Q(1,1) + f(0,0) = 1 + 0 = 1$, where $Q(1,1) = 1$ since there is only 1 query (i.e., q_4) that intersects interval I_1 and does not intersect interval I_2. Thus $f(1,1) = min(1,2) = 1$. Take the entry $(2,3)$ as another example. The PUSH value is $U(2) + f(1,3) = 1 + 2 = 3$. The PULL value is $Q(2,3) + f(1,1) = 3 + 1 = 4$, where $Q(2,3) = 3$ since there are 3 queries (q_2, q_5, and q_6) that intersect interval I_2 and do not intersect interval I_4. Thus $f(2,3) = min(3,4) = 3$.

In general, in order to find an optimal partition for the original problem $P(n,n)$, we fill the entries of an $n \times n$ table. We fill all rows for a particular column, then move to the next column. For each entry (i,j), we compute the PUSH value, the PULL value, and the $f(i,j)$ value using the recurrence function. We repeat the process until we fill the entry (n,n). Then we backtrack the computation process to identify the labels for the intervals in order to achieve this $f(n,n)$ value. In particular, if the minimum value for $f(n,n)$ is when it is in the PUSH mode, in the optimal solution for the original problem, interval I_n is in PUSH. In that case, we next look at the entry for $f(n-1,n)$. Again, depending on in which mode it has the minimum value, we decide the mode for interval I_{n-1} in the optimal partition. On the other hand, if the minimum value for $f(n,n)$ is when it is in PULL, we label I_n as PULL, and look at the entry for $f(n-1,n-1)$. This process of moving in the reverse order along the intervals to determine their modes in the optimal solution is shown in Fig. 4 by the arrows beginning with $f(4,4)$. The final optimal partition is: I_2 is PUSH, and the other three intervals are PULL, as presented in Section 2. The optimal cost $f(4,4)$ is 5.

3.3.2 Correctness Proof

We give a correctness proof of the DYNPROG algorithm. Let $\mathbb{U}$ denote the given set of updates and $\mathbb{Q}$ denote the given set of queries in the n intervals $I_1, \ldots, I_n$. Let $\mathbb{U}(i)$ denote the subset of updates in intervals I_l, where $1 \leq l \leq i$, and $\mathbb{Q}(j)$ denote the subset of queries that do not intersect any interval I_l, where $(j+1) \leq l \leq n$. Note that $\mathbb{U} = \mathbb{U}(n)$ and $\mathbb{Q} = \mathbb{Q}(n)$. Let $\Pi(\mathbb{U}, \mathbb{Q})$ denote the gerrymandering problem we are trying to solve, and $\Pi(\mathbb{U}(i), \mathbb{Q}(j))$ the corresponding problem on the subsets $\mathbb{U}(i)$ and $\mathbb{Q}(j)$. We shall use $\Pi(i, j)$ as short for $\Pi(\mathbb{U}(i), \mathbb{Q}(j))$, and $c(i, j)$ as the optimal cost for $\Pi(i, j)$. Note that, in $\Pi(i, j)$, the intervals $I_{i+1}, \ldots, I_n$ can be (trivially) labeled PUSH as there are no updates in those intervals.

A *constrained* problem is one in which an additional constraint has to be satisfied: Given $\Pi(\mathbb{U}, \mathbb{Q})$, the problem $\Pi(\mathbb{U}, \mathbb{Q}; C)$ is a constrained problem in which the solution has to satisfy the predicate C. E.g., $\Pi(\mathbb{U}, \mathbb{Q}; I_i \in \text{PUSH})$ is the problem in which the solution is the minimum cost partition that labels interval I_i PUSH. Define analogously, $\Pi(i, j; C)$ and $c(i, j; C)$.

In the following we represent a partition of the intervals by $\mathcal{A}$, a sequence of ordered pairs (I_i, S_i), where $S_i \in \{\text{PUSH}, \text{PULL}\}$, for $1 \leq i \leq n$. We represent a labelling for $\Pi(i, j)$ by a sequence of i ordered pairs, since the rest ($(i + 1)$ onwards) are assumed to be labeled PUSH.

As before, *pull queries* are queries that intersect intervals labeled PULL and *push queries* are the remaining queries. Also, the cost of a PUSH interval is the cost of its updates. The proof of correctness of the dynamic program follows from the following Lemma 3.1 and Lemma 3.2.

Lemma 3.1 $c(i, j) = \min\{c(i, j; I_i \in \textit{PUSH}), c(i, j; I_i \in \textit{PULL})\}$. □

Proof: The optimal solution for $\Pi(i, j)$ labels I_j one of PUSH or PULL. ■

Lemma 3.2 *For* $i \leq j$, $c(i, j; I_i \in \textit{PUSH}) = U(i) + c(i-1, j)$ *and* $c(i, j; I_i \in \textit{PULL}) = Q(i, j) + c(i-1, i-1)$. □

Proof: Consider the problem $\Pi(i, j; I_i \in \text{PUSH})$. Let $\mathcal{A}'$ be the optimal partition for $\Pi(i - 1, j)$. Now, $\mathcal{A} = \mathcal{A}' \cup (I_i, \text{PUSH})$ is a partition for $\Pi(i, j; I_i \in \text{PUSH})$. We claim that the cost $c(\mathcal{A})$ is $c(\mathcal{A}') + U(i)$. To see this, first consider the updates: A PUSH interval in $\mathcal{A}$ is either a PUSH interval in $\mathcal{A}'$ or is I_i. In the former case its update cost is included in the term $c(\mathcal{A}')$, and in the latter case the update cost is $U(i)$. Now consider the pull queries. Every query in $\mathbb{Q}(j)$ that is a pull query according to $\mathcal{A}$, is a pull query according to $\mathcal{A}'$; since the interval I_i is a PUSH interval. So the cost of all pull queries in $\mathcal{A}$ is included in the term $c(\mathcal{A}')$. This proves our claim. We next show that $\mathcal{A}$ is an optimal solution for $\Pi(i, j; I_i \in \text{PUSH})$. For proof by contradiction, assume that there exists a partition $\mathcal{B}$ with strictly lower cost for $\Pi(i, j; I_i \in \text{PUSH})$. Clearly, $\mathcal{B}$ can be decomposed into $\mathcal{B}' \cup (I_i, \text{PUSH})$. Now $\mathcal{B}'$ is a partition for $\Pi(i - 1, j)$. As before, if $c(\mathcal{B}')$ is the cost of $\mathcal{B}'$, the cost $c(\mathcal{B})$ is $c(\mathcal{B}') + U(i)$. This implies that $c(\mathcal{B}')$ is strictly less than $c(\mathcal{A}')$, the optimal cost for $\Pi(i - 1, j)$. This is a contradiction.

Now consider the problem $\Pi(i, j; I_i \in \text{PULL})$. Let $\mathcal{A}'$ be the optimal partition for the problem $\Pi(i-1, i-1)$. Now, $\mathcal{A} = \mathcal{A}' \cup (I_i, \text{PULL})$ is a partition for $\Pi(i, j; I_i \in \text{PULL})$. We claim that the cost $c(\mathcal{A})$ is $c(\mathcal{A}') + Q(i, j)$. To see this, first consider the updates. For every PUSH interval in $\mathcal{A}$ there is a corresponding PUSH interval in $\mathcal{A}'$; since I_i is a PULL interval. So its cost is included in the term $c(\mathcal{A}')$. Now consider pull queries. For every pull query in $\mathcal{A}$ there is either a corresponding pull query in $\mathcal{A}'$ or the query intersects I_i, but not both (since no query in $\mathbb{Q}(i - 1)$ intersects I_i, and all the intervals $I_{i+1}, \ldots, I_j$ are marked PUSH). In the former case its cost in included in the term $c(\mathcal{A}')$ and in the latter case its cost is included in the term $Q(i, j)$. This proves our claim. To show that $\mathcal{A}$ is the optimal partition for $\Pi(i, j; I_i \in \text{PULL})$, assume for a proof by contradiction that there exists a partition $\mathcal{B}$ with strictly lower cost. Again, as before, $\mathcal{B}$ can be decomposed into $\mathcal{B}' \cup (I_i, \text{PULL})$. Further, $\mathcal{B}'$ is a partition for $\Pi(i-1, i-1)$. It also follows that, if $c(\mathcal{B}')$ is the cost of $\mathcal{B}'$, the cost $c(\mathcal{B})$ is $c(\mathcal{B}') + Q(i, j)$. This implies that $c(\mathcal{B}')$ is strictly less than $c(\mathcal{A}')$, the optimal cost for $\Pi(i - 1, i - 1)$. This is a contradiction. ■

3.3.3 Computing $Q(i,j)$ Values Efficiently

Let l be the maximum number of intervals intersected by any single query. We compute the $Q(i,j)$ values in n phases. Begin with a sorted sequence of the starting points of the queries. In phase i, we compute the $Q(i,j)$ values ($j = i, \ldots, i+l-1$). Through the different phases we maintain an array R of n values. At the beginning of phase i, for every $j \geq i$, $R(j)$ is the number of queries that start in an interval before I_i but end at interval I_j. At the beginning of phase 1, all values of $R(j)$ are 0.

Phase Number	R(1)	R(2)	R(3)	R(4)
1	1	0	1	▨
2	▨	2	1	1
3	▨	▨	1	1
4	▨	▨	▨	2

Figure 5: Computing $Q(2,4)$.

In phase i, look at the sorted sequence of queries and for each query $Q(i,j)$ that starts in interval I_i and ends in some interval I_j, add 1 to $R(j)$. Now $R(j)$ is the number of queries that start in interval I_i or earlier, but end at interval I_j. Note that $Q(i,i)$ is the value $R(i)$. Similarly, $Q(i,i+1)$ is the value $Q(i,i) + R(i+1)$. In general, $Q(i,j)$ is the value $Q(i,j-1) + R(j)$. Thus, by a single pass through l values of R, we can compute all the $Q(i,j)$ values for this phase. After this phase, we move to phase $(i+1)$. Fig. 5 shows a run of this computation on the example given in Fig. 1.

3.3.4 Complexity

Let n be the number of intervals of interest, l be the maximum number of intervals intersected by any single query. We denote the number of queries in the workload as m_q, and the number of updates m_u. Therefore, the size of the input is $m = m_q + m_u$. It is easy to see that the space complexity of the algorithm is the size of the table for $f(i,j)$, which is $O(nl)$, plus the space complexity $O(nl)$ to compute and store the $Q(i,j)$ values. For the time complexity, most tasks such as computing intervals of interest take $O(m \lg m)$ time. In computing the $Q(i,j)$ values, each query $Q(i,j)$ is considered only once, which takes $O(m_q)$ time. For each of the n phases, l values of R are looked up in $O(nl)$ time. The time to fill the entries of the table for $f(i,j)$ takes $O(nl)$ time. The relationship between m_q and n, based on geometry, is $n+1 \leq 2m_q \leq n(n-1)$. Therefore, we have:

Theorem 3.1 *Algorithm* DYNPROG *runs in $O(nl+m)$ time and takes $O(nl)$ space.* □

3.4 Model $\mathcal{M}_2$: Constant Cost

We next consider the cost model $\mathcal{M}_2$, which generalizes model $\mathcal{M}_1$ by allowing each event (update or query) to have a constant cost.

Cost model $\mathcal{M}_2$: Each communication between client and server has constant cost. Different events can have different costs.

This model allows the cases where the costs associated with queries and updates are not necessarily the same. The sources of this cost asymmetry can be various. For instance, it is possible that the cost of pulling a query costs more than pushing an update as is seen in mobile networks [Bar99], where the mobile agents have limited power compared to their base station.

Notice that in this model, the cost of pulling a query does not depend on the PUSH/PULL partition of the domain. That is, even though the client has cached up-to-date objects for some PUSH intervals, what objects have been cached does not affect the cost of pulling a query. This assumption is true for queries whose communication cost is not very related to the size of data, such as aggregation queries. As an example, to answer an aggregation query such as "Finding the number of cars in the price range $5,000 and $10,000," the server only needs to send a number to the client, independent from what objects have been cached. This assumption is also valid for cases where the communication cost is mainly determined by initial setup and transmission delay, and is not related to the cached objects.

Under this model $\mathcal{M}_2$, given a workload W, let $c(u_i)$ be the cost of an update u_i in W under this model, and $c(q_j)$ be the cost of pulling a query q_j in W. We modify the DYNPROG algorithm to find an optimal labeled partition by using the following recurrence instead:

$$\begin{aligned} i > 0 : f(i,j) &= \min \begin{cases} \mathcal{U}(i) + f(i-1,j) & \text{PUSH} \\ \mathcal{Q}(i,j) + f(i-1,i-1) & \text{PULL} \end{cases} \\ f(0,j) &= 0. \end{aligned}$$

Here $\mathcal{U}(i)$ is the total cost for all the updates in interval I_i, and $\mathcal{Q}(i,j)$ is the total cost for all queries intersecting interval I_i but do not intersect interval I_{j+1}. We can show that this modified algorithm can find an optimal solution under this cost model.

3.5 Model $\mathcal{M}_3$: Linear Communication Cost

Under this cost model, each communication is linear to the size of transferred data.

Cost model $\mathcal{M}_3$: The cost of transferring data of size s is $\alpha + \beta \times s$, where α and β are constants.

Since each pushed update needs to send the updated data to the client, the cost of such a pushed update is a constant, assuming all the updated objects have the same size. Thus we can compute the costs for pushed updates. For each pulled query, its cost depends on the PUSH/PULL labels for the intervals of interests. For instance, consider the query q_2 in Fig. 1. If we mark I_1 as a PULL interval, then this query needs to be pulled. However, since the cost of pulling this query depends on the data size, the cost is also related to how other intervals are labeled. If interval I_2 is labeled PUSH, then the server does not need to send any data in I_2, since the data has been cached up-to-date at the client. If interval I_2 is labeled PULL, then the server still needs to send the data in this interval to the client. In summary, the cost of a pulled query is no longer a constant, since it depends on the PUSH/PULL labeling of the domain.

Suppose the number of objects within each interval I_i does not change much during the sequence of the workload, and its size is assumed to be a constant $N(i)$ in the workload. Now we show how to modify the DYNPROG algorithm slightly to find an optimal solution under this cost model $\mathcal{M}_3$. In the recurrence function, the cost for the PUSH case should be:

$$\mathcal{U}(i) + f(i-1,j),$$

where $\mathcal{U}(i)$ is the total cost for all the updates in interval I_i. The cost for the PULL case should become:

$$\alpha \times Q(i,j) + \beta \times N(i) \times Q(i) + f(i-1,i-1).$$

The rationale behind this formula is the following. If we mark interval I_i PULL, we need to pay the "α" portion (in the cost model) for those queries $Q(i,j)$ intersecting I_i but *not* intersecting interval I_{j+1}, i.e., $\alpha \times Q(i,j)$.

In other words, we pay these costs for these queries only once. In addition, all the queries intersecting this interval need to pay the cost for transferring the data of size $N(i)$ in this interval, corresponding to the "$\beta \times s$" portion in the cost model. This total cost is computed as $\beta \times N(i) \times Q(i)$, where $Q(i)$ is the number of queries intersecting this interval I_i.

In summary, the DYNPROG algorithm can be extended to various communication cost models to find an optimal labeled partition for a given workload.

4 Efficient Gerrymandering

In many cases we need to find a labeled partition efficiently in both time and space, especially when we need to recompute such a partition in the presence of changing workloads (Section 5). In these cases, we need to consider other, more efficient, heuristics to find a good labeled partition. In this section, we first study how to do efficient gerrymandering for a single attribute given a workload. We develop a family of efficient heuristics. We then extend them to cases of multiple gerrymandering attributes, and discuss how to choose gerrymandering attributes. For simplicity we assume the communication cost model $\mathcal{M}_1$, and our results can be extended to other cost models.

4.1 Efficient Gerrymandering on a Single Attribute

4.1.1 Efficient Partitioning

Given a workload and a single gerrymandering attribute, we want to find a good labeled partition quickly. In developing heuristics for finding such a solution, the first problem we need to consider is how to partition the domain of the attribute into intervals. One approach is to use those intervals of interest formed by those starting points and ending points of the queries in the workload, as defined in Section 3.2.

Another approach is to divide the entire domain of the attribute into B intervals of equal width, called "buckets," where B is a parameter to be tuned. This approach has the flavor of equi-width histograms [PIHS96], which split the range into uniformly-sized bucket and study the properties of the buckets. One advantage of this approach is that the storage space required to store the gerrymandering information and the corresponding running time are bounded in size to $O(B)$, which could be independent from the workload. We could also adopt the idea of equi-height histograms [PIHS96] to decide intervals, which requires a function to measure the "volume" of each bucket, e.g., by using the total number of queries and updates in the bucket.

4.1.2 Efficient Labeling

After deciding the intervals, we need to choose a PUSH/PULL label for each interval. We present a family of heuristics for finding a (possibly suboptimal) labeled partition efficiently, assuming the intervals have been decided. We will use the workload in Fig. 6 to illustrate these heuristics. The figure shows numbers of queries for ranges. For instance, there are 7 queries with a range condition $(8, 14)$. There are 10 intervals of interest $I_1, \ldots, I_{10}$. The figure also shows the number of updates for each I_i, e.g., there are 5 point updates in interval $I_3 = (8, 9)$. We use heuristics to decide the PUSH/PULL labels for those intervals formed by those query starting points and ending points.

NAIVE: This heuristic is similar to the idea proposed in [KB96]. It makes a PUSH/PULL decision for each interval locally, as if the interval were the entire range. That is, for each interval I_j, if the number of queries intersecting this interval (i.e., $Q(j)$) is greater than the number of updates in the interval (i.e., $U(j)$), we label I_j as PUSH; otherwise, we label it as PULL. Formally,

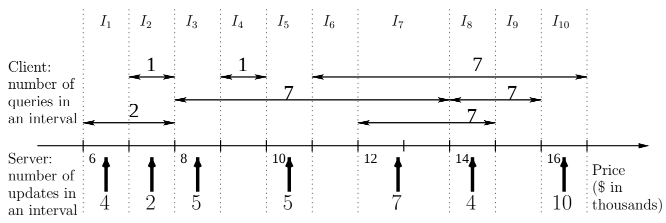


Figure 6: A workload for one range attribute.

```
NAIVE: if $U(j) > Q(i)$ : $I_{\text{PULL}} \leftarrow i$
else $I_{\text{PUSH}} \leftarrow i$
```

In Section 6 we show that NAIVE proves to be a good heuristic in practice with low running times and cost not much more than the optimal on both real and synthetic data sets. For synthetic data sets it outperforms all heuristics except it's cousin MNAIVE described next in all cases except when the number of updates are high.

MNAIVE: It goes through the intervals from I_1 to I_n, marking them PUSH or PULL as NAIVE, except that whenever an interval gets marked PULL, all the queries intersecting that interval are removed and not considered again for any future labeling decision. The intuition is that a query passing through a PULL region is already "paid for," and can be ignored when we consider other later intervals. In Section 6 we show that MNAIVE is, like NAIVE, very good in terms of both running time and communication cost in all cases except when the number of updates is high. Besides it consistently outperforms NAIVE in all cases.

RANDOMIZE: Here we randomly decide to mark interval i PUSH with probability proportional to $w(i)$ and PULL with probability proportional to $U(j)$. Formally,

```
RANDOMIZE: with probability $\frac{U(j)}{w(i)+U(j)}$ : $I_{\text{PULL}} \leftarrow i$
with probability $\frac{w(i)}{w(i)+U(j)}$ : $I_{\text{PUSH}} \leftarrow i$
```

Note that in this method we also apply the modification used in MNAIVE, i.e. queries intersecting a region marked PULL get removed from the input and do not affect future decisions. In Section 6 we see that in experimental settings this method is much more communication efficient than MNAIVE or NAIVE when the number of updates is large.

PROP: This heuristic differs from the previous ones as follows. Instead of counting a query as one unit at each intersecting interval, we divide the cost of the query amongst these intervals proportionally to the lengths of these interval. Let $w(j)$ represent the aggregated cost of all queries intersecting interval I_j. This heuristic makes a deterministic local decision for interval I_j based on the distributed query cost $w(j)$ and update cost $U(j)$ as follows: if $U(j) > w(j)$, we label I_j as PULL; otherwise, we label it as PUSH. It also applies the modification used in MNAIVE, i.e., queries intersecting a region already marked PULL get removed from the input and do not affect future decisions. A variation of this heuristic, called RANDOMIZE, randomly decides to mark interval I_j PUSH with probability $\frac{w(j)}{w(j)+U(j)}$ and PULL with probability $\frac{U(j)}{w(j)+U(j)}$.

Like RANDOMIZE, this method is seen to be much more communication efficient than the others when the number of updates is high (cf. Section 6.) We denote the number of intervals intersected by a query q as $l(q)$. Formally, we calculate a query weight function $w(j)$ for an interval i as $w(j) = \sum_{q \text{ overlaps } I_x} \frac{1}{l(q)}$. The algorithm assigns a label as follows.

```
PROP: if $U(j) > w(j)$ : $I_{\text{PULL}} \leftarrow i$
else $I_{\text{PUSH}} \leftarrow i$
```

Heuristics	(6,7)	(7,8)	(8,9)	(9,10)	(10,11)	(11,12)	(12,14)	(14,15)	(15,16)	(16,17)	$Cost_U$	$Cost_Q$	Total
1. NAIVE	L	H	H	H	H	H	H	H	H	L	23	9	32
2. MNAIVE	L	L	H	H	H	H	H	H	H	L	21	10	31
3. PROP	L	L	L	H	L	H	L	L	H	L	0	31	31
4. DYNPROG	L	L	L	H	L	H	H	H	H	L	11	17	28
5. BUCKETS	L	L	H	H	H	H	H	H	H	H	31	3	34

Table 1: Labeled partitions by different heuristics for the workload in Fig. 6 ("H" for "PUSH" and "L" for "PULL"). The first four heuristics are using the intervals formed by the starting/ending points in the query conditions. The last one, called BUCKETS, is running NAIVE on $B = 5$ equi-width buckets.

Table 1 shows the results of running these heuristics (including the DYNPROG algorithm) on the workload in Fig. 6. Among these heuristics, MNAIVE and PROP achieve the lowest cost 31. DYNPROG computes an optimal labeled partition. Notice that these heuristics can be used with other ways to partition the domain into intervals. For instance, the figure also shows the result of an heuristic, called BUCKETS, which runs NAIVE on $B = 5$ equi-width buckets.

In general, which heuristic to use depends on factors such as the workload, how often the heuristic needs to be run, and how much requirement the application has on the time and space efficiency. For instance, if the application does not run the heuristic very often, and can accept a long running time, the DYNPROG algorithm could be a good choice. Otherwise, we can choose one of the heuristics.

4.1.3 Pruning Queries and Updates

We present two simple rules that help prune a problem instance and can be used as pre-processing steps before running a heuristic. Depending on the number of queries and updates pruned, this could contribute significantly in reducing the running time.

- **Rule 1**: If for an interval i the number of updates $U(j)$ is larger than the number of queries intersecting it $Q(i)$, then the interval is assigned PULL, and the updates in it can be pruned.
- **Rule 2**: If for an interval i the number of updates $U(j)$ is less than the number of queries *contained* in the interval, then the interval is assigned PUSH, and the queries intersecting interval i and the updates in it can be removed.

Note that when pruning queries or updates to get a reduced problem, the cost of the original problem is the sum of the cost of the reduced problem and pruned pushed updates, or pruned pulled queries.

4.2 Efficient Gerrymandering on Multiple Attributes

When we use gerrymandering on multiple attributes, the number of semantic regions for the entire space could be larger than that of a single attribute. Thus it becomes more important to find a good labeled partition efficiently for a given workload to reduce communication cost. Similarly to the one-attribute case, we need to first decide a partition for each attribute. The following claim extends Proposition 3.1.

Proposition 4.1 *Given a workload, let $p_{j,1} < p_{j,2} < \ldots < p_{j,n_j}$ be the starting and ending points of the ranges in the query conditions for attribute A_j, where $p_{j,1}$ could be $-\infty$, and p_{j,n_j} could be $+\infty$. Under any communication cost model, there is an optimal labeled partition whose regions have the following property. For each region, its interval for each attribute A_j is either of the form $(p_{j,i}, p_{j,i+1})$, or the form $[p_{j,i}, p_{j,i}]$.* $\Box$

Unfortunately, a natural extension of the DYNPROG algorithm to multiple attributes results in an exponential-time algorithm. Thus it is appealing to extend those heuristics due to their simplicity and better efficiency. Before using these heuristics, we need to decide a partition for the semantic space. As in the single-attribute case, we can use the partition based on the starting/ending points of queries in the workload on the gerrymandering attributes. We can also consider a partition formed by equi-width buckets for each attributes. Other methods are also possible to generate a partition.

It is straightforward to extend the heuristics to decide PUSH/PULL labels for the semantic regions in a decided partition. Take the MNAIVE heuristic as an example. In the multi-attribute case, we go through the semantic regions following a certain order. For each region, we compute the total cost $cost_u$ for the updates in the region, and the total cost $cost_q$ for the queries intersecting this region. If $cost_u < cost_q$, we mark this region PUSH; otherwise, we mark it PULL, and ignore all queries intersecting this region when considering later regions. We can choose any order to go through the regions, e.g., by using an increasing order based on the total cost of intersecting queries for each region.

4.3 Choosing Gerrymandering Attributes

Since gerrymandering is used on a set of attributes, the client and the server can choose the gerrymandering attributes based on (1) the saving on the communication cost provided by these attributes; and (2) the time/space efficiency to do gerrymandering on these attributes. One way to choose such attributes is as follows. Given a workload, we analyze the conditions in the queries. We choose attributes on which many queries have specified conditions. These attributes could be used for gerrymandering, since each such query with conditions on these attributes is likely to intersect PUSH regions, and the client does not need to contact the server to answer the query. However, we may only need a subset of these candidate attributes for gerrymandering.

One greedy approach to choosing such a subset is the following. For each attribute, we run one of the heuristics above or DYNPROG to find a labeled partition for the given workload, and compute the corresponding total communication cost. We choose the attribute A_1 with the minimum total cost as one gerrymandering attribute. We then add another candidate attribute to A_1, and run an heuristic on both attributes to find a labeled partition and compute the total communication cost. We choose the attribute A_2 such that gerrymandering on $\{A_1, A_2\}$ has the minimum communication cost. We repeat this process until we use all the candidate attributes for gerrymandering, the communication cost does not decrease much, or computation becomes too expensive.

5 Gerrymandering for Dynamically Evolving Workloads

In this section we study how to gerrymander the range in the presence of dynamic or evolving workloads of queries and updates. There are many situations where the techniques developed so far will serve us adequately for the case where workloads change. For instance, many client-server applications see queries and updates whose workload tends to be repetitive. In this case, we can use an earlier workload to find a labeled partition, and use the partition to do gerrymandering for a time period in which the workload is expected to be similar. In our running example of the online car shop, after analyzing the weekly queries on the client queries and data updates on the server, its web master finds that the workloads of different weeks are similar. One reason could be due to the fact that car buyers tend to do research on cars by issuing queries throughout the week, while many cars can be added or sold on weekends, causing a lot of data updates on the server. In this case, the web master can use the workload of one week to decide a labeled partition, and use it for every week; and correspondingly use the workload from one weekend as a template for all the weekends in future.

Challenges arise when past input patterns do not contain enough information about the future, i.e., when the patterns of inputs change with time. A natural and efficient approach to the dynamic setting is to extend the heuristics we gave in Section 4.1. Consider, for example, the heuristic NAIVE. The arrival of a new query or update means that some interval or region might shift from having more queries to having more updates or vice versa. If this happens, we may switch the label of this region.

The dynamic program is computationally expensive. Rerunning it every time a new input comes is neither feasible nor necessary. We need to detect substantial changes in the distribution of the queries and updates, and recompute a new labeled partition by running the dynamic program only if necessary. Two questions need to be answered. (1) When can we say that the workload has changed significantly enough to warrant recomputing a labeled partition? (2) How do we efficiently compute a new labeled partition?

5.1 Detecting Changes in Workload Patterns

We first develop criteria for determining when the pattern of the workload has changed enough to warrant a recomputation of our current labeling. We give three heuristics for this purpose. Each of these heuristics is valid for all three cost models discussed in Section 3, but we present them in terms of $\mathcal{M}_1$. The first two heuristics, DYNNAIVE and DYNPROP, take local views of the intervals and trigger a recomputation when a large number of intervals appear to be incorrectly labeled. The third one, DENSITY, takes the communication cost per input as the criterion and demands recomputation when this density shows signs of growing beyond error. We discuss here the recomputation trigger mechanisms for these heuristics, postponing the discussion of how they relabel.

DYNNAIVE. Intuitively, the idea behind this method is that a PUSH interval should be numerically dominated by queries and hence it should be relabeled when this domination recedes. Symmetrically a PULL interval needs to be relabeled when the domination of the number of updates is in decline. To realize this intuition formally, we begin with a workload of queries and updates X and run the dynamic program on it obtaining a solution (I_X, S_X), where I_X is a set of intervals and S_X, the partition, is a function from I_X to $\{\text{PUSH}, \text{PULL}\}$. For each interval $i \in I_X$ we compute its *base ratio*, $\gamma_X(i)$ as follows:

$$\gamma_X(i) = \begin{cases} \mathcal{Q}(i, |I_X|)/\mathcal{U}(i) & \text{if } S_X(i) = \text{PUSH} \\ \mathcal{U}(i)/Q(i, |I_X|) & \text{if } S_X(i) = \text{PULL} \end{cases}$$

The base ratio for a “good” interval i should be greater than 1 and the larger it is, the less is the cost incurred by the interval i. It gives us a base case against which to measure the performance of the interval. If in the future the value of the ratio drops significantly below its original value, it indicates that i is now “bad” (incorrectly labeled).

Suppose we denote the input stream since the dynamic program was run on X by Y. We compute $\gamma_Y(i)$ by considering the labeling S_X but computing the ratios *only* for the queries and updates which are part of Y. We trigger a recomputation if too many intervals have become bad. Formally, the recomputation is triggered based on two parameters $\alpha, \beta < 1$ as follows:

A new labeling is required if at least $\beta \cdot n$ intervals have $\gamma_Y(i) \leq \alpha \cdot \gamma_X(i)$.

DYNPROP. This method has the same flavor as DYNNAIVE except that the way in which we count queries is different. We distribute the weight of each query evenly over all the intervals it intersects. Instead of using the number of queries to compute the base ratio of interval i, we use the sum of the weights contributed by each query to interval i. Compared to DYNNAIVE, DYNPROP tends to assign lower values to base ratios of PUSHintervals, and higher values to those of PULLintervals.

Carrying forward the notation from the previous heuristic, we add: For a query $q \in X$, $h_X(q, i) = 1$ if q intersects interval i. Now we can define a query weight per interval as follows:

$$\mathcal{QW}(i) = \sum_{q \in X_q} \frac{h_X(q,i)}{\sum_{i=1}^{|I_X|} h_X(q,i)}$$

Where $X_q \subseteq X$ is the set of all queries in X. This query weight function makes sure that if a query stretches across k intervals, each interval receives weight $1/k$ from it.

The base ratio for this heuristic is $\eta(i)$ defined as follows:

$$\eta_X(i) = \begin{cases} \mathcal{QW}(i)/\mathcal{U}(i) & \text{if } S_X(i) = \text{PUSH} \\ \mathcal{U}(i)/\mathcal{QW}(i) & \text{if } S_X(i) = \text{PULL} \end{cases}$$

As before, we recompute based on two parameters α, β:

A new labeling is required if at least $\beta \cdot n$ intervals have $\eta_Y(i) \leq \alpha \cdot \eta_X(i)$.

DENSITY. Given an input workload X, as before we compute a dynamic programming solution (I_X, S_X). The communication cost of this solution is the number of updates pushed plus the number of queries pulled (see Section 3.1). Denote it by $C(X, I_X, S_X)$.

$$\begin{aligned} C(X, I_X, S_X) &= |\{u \in X_u | u \in i, S_X(i) = \text{PUSH}\}| \\ &\quad + |\{q \in X_q | \exists i : q \in i, S_X(i) = \text{PULL}\}| \end{aligned}$$

For the subsequent input Y, we compute $C(Y, I_X, S_X)$, i.e., the communication cost of the queries and updates in Y under the labeling S_X in the interval partition I_X. Now we say that, given a parameter $\alpha > 1$, a new labeling is required if the cost density of the new queries and updates deviates from the cost density for the original input X. Formally, trigger a recomputation when

$$\frac{C(Y, I_X, S_X)}{|Y|} > \alpha \cdot \frac{C(X, I_X, S_X)}{|X|}.$$

The intuition is that the communication cost per item should stay low through the life of the system. If this cost density starts increasing, it indicates that we are using a faulty labeling.

5.2 Recomputing a Labeled Partition

Having decided that the current labeling is unsatisfactory, we have to recompute it. Note that here we continue to use the static algorithm described earlier, DYNPROG. We simply provide it with a new workload. A naive approach would be to give it as input all the queries and updates seen so far. But running the dynamic program again on the old input and the new input together is inefficient and unnecessary. We propose two reductions to provide a smaller workload which is additionally more relevant to the evolving communication scenario.

Removing redundant items. We claim that all updates and queries in the old workload X which were in PUSH regions can be removed. The intuition is that, once an update has been pushed, the communication cost for it has been paid already. Thus removing it and all queries which overlap it will start off that interval as a clean slate, making it less likely for new updates to be pushed (unless they get a large number of queries to justify the pushing). We ensure that queries overlapping with some PULL regions are not entirely removed. They are only removed from PUSH regions, and might continue into the next run of the dynamic program as non-contiguous entities. That is, we might get a single query comprising a number of non-contiguous fragments.

A windowing approach. Having removed the redundant items as above we can additionally keep a parameter L that limits the number of queries and updates used to recompute the labeling. That is, when we recompute the labeling, we only use the last L queries and updates at the server, having removed items from before the previous run which got labeled PUSH. The advantage of this approach is that the labeling is more reflective of the recent events. Additionally the running time of each run remains roughly the same. The actual cost incurred in relabeling regions might involve updates which were not included in the window. This is so because if a region was earlier marked PULL and is now relabeled PUSH, all the updates from the beginning (possibly before the window began) that have not been pushed, have to be now pushed.

6 Experiments

In this section, we present experimental results that demonstrate the usefulness of data gerrymandering, and compare the performance of the various approaches we presented in the earlier sections for different scenarios. We simulated a client-server environment by considering workloads of client queries and server data updates. We used real updates and queries from the logs of the Sloan Digital Sky Survey (SDSS) [sdsa]. We also used synthetic workloads generated with controlled variations in number of queries, number of updates, query lengths, etc., to evaluate our algorithms under different types of inputs. Using these workloads we tested our methods in simulated client-server environments. We measured the communication costs under the three cost models in Section 3. In particular, once an algorithm or heuristic produced a labeled partition for a workload, we added the cost of each pulled query and each pushed update, where the costs were based on the cost model. In addition, we measured the actual running time for each algorithm.

All our experiments were implemented in C compiled using the GNU C compiler. We ran the experiments on a machine with four Pentium PIII Xeon 500MHz processors with 512KB cache each and overall memory of 4 GB, and a Linux operating system (except where specified; as in Section 6.1.1). In the following we present our results and give analytical explanations for the different behavior of the algorithms.

6.1 Static Workloads

We implemented the optimal algorithm and the heuristics discussed in Sections 3 and 4: DYNPROG (which gives the optimal labeled partition), UNIFORM (in which a single PUSH or a PULL decision is made uniformly for the entire domain, based on which is cheaper), NAIVE, MNAIVE, PROP, RANDOMIZE, and BUCKETS (with $B = 500$ buckets). We first focused on the case with a single range attribute under the communication cost model $\mathcal{M}_1$, i.e., each pushed update or pulled query has the same cost. In these algorithms, UNIFORM can be looked upon as an approach without Data Gerrymandering, and thus serves as a baseline to compare other algorithms with. Its cost depends only on the relative costs of queries and updates, and not on their distributions along the range attribute. We ran the algorithms above on real workloads from the SDSS as well as on synthetic workloads. For the real workload the emphasis was on verifying the usefulness of gerrymandering. For the synthetic workload, the emphasis was a detailed evaluation of our algorithms in various scenarios.

6.1.1 Real Static Workloads

For real-life workloads we used the database of the SDSS, in which objects are heavenly bodies such as stars and galaxies, and have over 400 attributes, including light intensities in various wavelengths, spectral data, measurement parameters, and, importantly, a two-dimensional position coordinate called "(ra, dec) values." Many of the queries received by the SDSS are range queries on both or one of these coordinates. Further, as the SDSS telescopes cover new regions in the sky, new objects are inserted into the database. Moreover, as the measurement and pre-processing techniques undergo improvements and modifications, old values are updated.

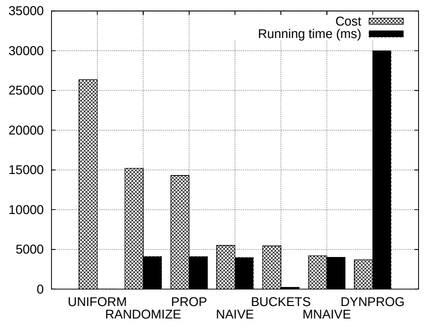


Figure 7: Communication costs and running times for a real SDSS workload.

For our real workload we picked a selection of 30,000 range queries from the query logs of Data Release 1 of the SDSS [sdsb]. For updates, we used a selection of 50,000 updates in the database between its Early Data Release and the Data Release 1. In Fig. 7 we show the communication costs and running times of the optimal DYNPROG and the heuristics. The machine used here (different from the rest of the experiments) had four 296 MHz UltraSPARC-II CPUs with 3 GB memory. Compared to the cost of the baseline UNIFORM, the cost of the optimal DYNPROG was less by a factor of more than 7. Thus, gerrymandering can indeed save on communication costs. The heuristic MNAIVE had a cost that was more than the optimal by a factor of just 1.13. Its running time was, however, less than that of DYNPROG by a factor of more than 7. Similarly, BUCKETS had a cost about a factor 1.47 more than the optimal, but a running time about 125 times less. Thus, for this workload, MNAIVE and BUCKETS were efficient heuristics.

6.1.2 Synthetic Static Workloads

For the synthetic workloads, we generated queries and updates as follows: The midpoints of queries and the updates formed clusters (usually 5 in number) on the range attribute; each cluster followed a Gaussian distribution, the mean and variance of which were chosen uniformly at random. The lengths of the queries followed a Gaussian distribution as well. We used well-known techniques [KMS00] to generate the Gaussians. We use $G(\mu, \sigma)$ to denote a Gaussian distribution with mean μ and standard deviation σ. Fig. 8 shows an example distribution of the updates and queries (using their middle points), in which the domain of the (range) attribute is the set of integers between $3,000$ and $20,000$. There were 5 clusters for the queries and 5 clusters for the updates. (We use this domain as an example, and any such a setting can be normalized to the range between 0 and 1, if preferred.)

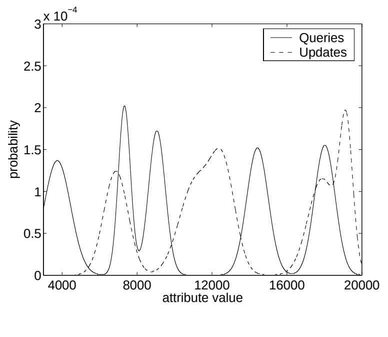
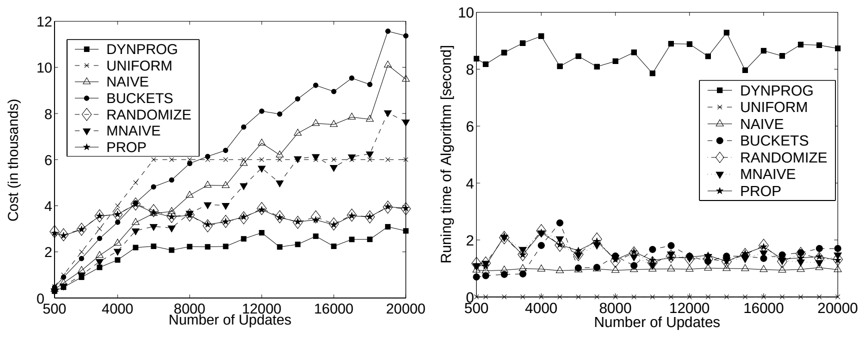


Figure 8: Example workload.

For each of the static-workload single-attribute algorithms, we measured the running time and the communication cost for the output labeled partition. In addition, we measured how these numbers changed for these algorithms as we increased the number of updates, increased the number of queries, changed the query lengths, and assigned different costs to the queries (for the communication cost model $\mathcal{M}_2$). For each setting, we ran the experiments on 100 different workloads, and computed the average (the results were very stable). We also measured the communication costs under cost model $\mathcal{M}_3$ and for the multi-attribute versions of these algorithms (see Section 4.2) on two-dimensional data.

Effect of Number of Updates. We first evaluated the algorithms as the number of updates changed. The attribute values were between $1,000$ and $30,000$.



(a) Communication cost ($\mathcal{M}_1$)

(b) Running time

Figure 9: Effect of number of updates. (6000 queries; query-length distribution: $G(500, 400)$.)

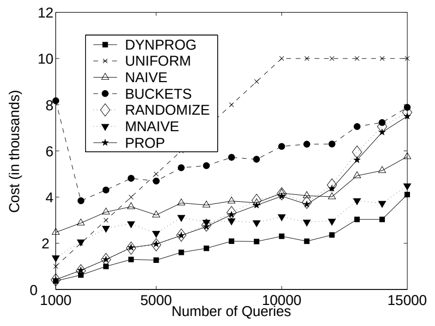
Fig. 9(a) shows the communication costs. As expected, DYNPROG always computed a labeled partition of minimum communication cost. Among the other algorithms, PROP was the best most of the time, while BUCKETS was the worst. When there were $8,000$ updates, the labeled partition generated by DYNPROG required about $3,000$ messages, while PROP required $4,200$ messages, and BUCKETS required more than $6,000$ messages. The baseline scheme, UNIFORM, required $8,000$ messages.

The "query-grabbing" heuristics, NAIVE, MNAIVE, and BUCKETS, performed poorly for large numbers of updates (greater than $16,000$), and had a cost greater than that of UNIFORM. This is not surprising: these heuristics disregard the possibility of a query overlapping several intervals — as if it occurs entirely within the current interval under consideration. This is equivalent to replacing a long query that covers r intervals with r single queries, one for each interval. As a result, these heuristics are biased towards assigning the PUSH mode for each interval. This tendency caused them to fail when the number of updates was large. In contrast, the "query-sharing" heuristics such as PROP and RANDOMIZE do not have a PUSH bias, and performed well for number of updates more than about $8,000$. Their cost was at most 1.5 times that of DYNPROG.

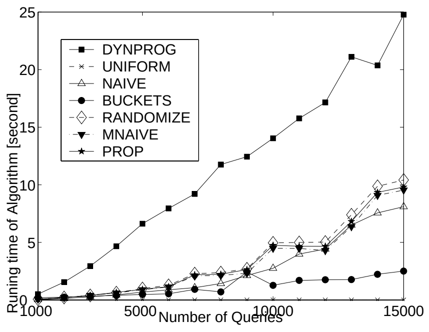
Query-sharing heuristics are, however, outperformed by the query-grabbing heuristics when the number of updates is small. Intuitively, this happens because a large number of intervals are nearly empty of updates. Sharing the weight of a query with such intervals is a "waste" and can lead to sub-optimal solutions. Query-grabbing heuristics do not have this problem. We observed this separation between the two types of heuristics in many experiments.

Fig. 9(b) shows the running times for different algorithms. All methods except DYNPROG and UNIFORM needed about the same amount of time, 1.5 seconds. DYNPROG took around 7.5 seconds, and UNIFORM required almost-zero time. Moreover, for all the algorithms, the running time did not change very much as we increased the number of updates. The reason is that the running time is mainly affected by the intervals created by the queries. Since the query workload was similar for different runs, the generated intervals were also similar.

Effect of Number of Queries. We evaluated the effect of the number of queries. The attribute values were between $1,000$ and $50,000$. Fig. 10(a) shows the communication costs for the labeled partitions generated by different algorithms. It highlights the power of gerrymandering. Consider the costs for $10,000$ queries. The baseline UNIFORM had a cost of $10,000$ messages, while the optimal DYNPROG had about $1,500$ messages. MNAIVE was the best among the heuristics when there were more than $8,000$ queries. At $10,000$ queries its cost was about $2,500$ messages. This figure also shows the difference in performance between query-grabbing algorithms and query-sharing algorithms based on the relative size of queries and updates (discussed in detail in the previous subsection).



(a) Communication cost ($\mathcal{M}_1$)



(b) Running time

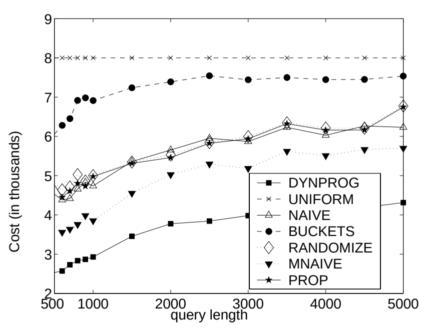
Figure 10: Effect of number of queries on communication cost. ($\mathcal{M}_1$; 10,000 updates; query-length distribution: $G(500, 400)$.)

Fig. 10(b) shows the running times of the algorithms. The time of all of them (except UNIFORM) grew linearly as the number of queries increased. The reason is that the complexity of most of them is closely related to the number of intervals of interest. More queries can create more intervals (close to linearly), causing the running time to increase. Among all these methods, DYNPROG required the most amount of time. For instance, when there were $10,000$ queries, it took DYNPROG about 14 seconds, while the other methods took less than 5 seconds. This figure, when contrasted with Fig. 9(a), shows clearly that the running time is affected more by a change in the number of queries, rather than a change in the number of updates.

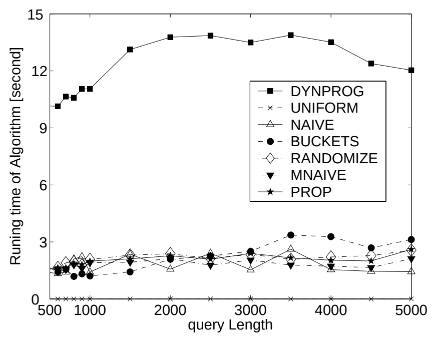
Effect of Query Lengths. The next set of experiments were to evaluate the effect of the query lengths on the performance of different methods. The range attribute values were between $1,000$ and $30,000$. The number of updates, as well as the number of queries, were fixed at $8,000$. We let the query lengths vary from 100 to $5,000$. The results are shown in Fig. 11. This experiment too shows differences between the various heuristics. When the query lengths increased, each query overlapped with more intervals (remember that the number of queries remained fixed). The query-grabbing NAIVE and BUCKETS performed poorly, mostly because they do not take into account that a query may overlap several intervals. Although RANDOMIZE, MNAIVE, and PROP also make local PUSH/PULL decisions, since all three remove queries that overlap an interval labeled PULL before proceeding, they produced better results. DYNPROG outperforms all others in terms of the communication cost but has the longest running time.

Effect of Query Costs under Model $\mathcal{M}_2$. Next we considered cost model $\mathcal{M}_2$, under which queries and updates can have different constant costs, and studied the DYNPROGRM and the heuristics which we modified suitably for this model. We assume each update has unit cost. For each query, we let its cost be uniformly distributed (at random) between 1 to 20. In this set of experiments, the range attribute values were between $1,000$ and $30,000$. The results are shown in Fig. 12.

Previously RANDOMIZE and PROP produced similar communication costs and running times. This was

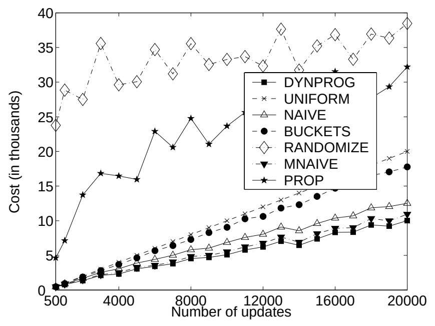


(a) Communication cost

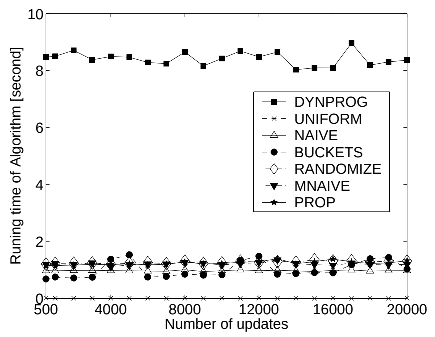


(b) Running time

Figure 11: Effect of query lengths on communication cost. ($\mathcal{M}_1$; $8,000$ updates; $8,000$ queries.)



(a) Communication cost



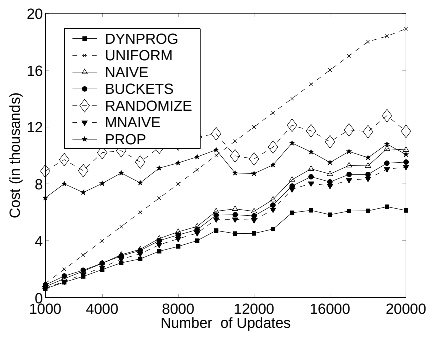
(b) Running time

Figure 12: Effect of query weights on communication cost. ($\mathcal{M}_2$; $6,000$ queries; query-length distribution: $G(500, 400)$.)

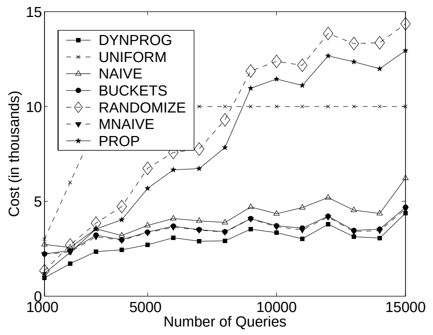
because in all the experiments discussed so far the query weights and update weights were roughly comparable. If the query weight is much larger than the update weight, almost all intervals should be marked PUSH. PROP does this deterministically. However, RANDOMIZE incorrectly marks intervals PULL with some probability. And, once it does so, since we remove pulled queries from the rest of the input, a snowball effect happens, with subsequent intervals now becoming more likely to get marked PULL incorrectly. In this experiment, because the queries are weighted, the ratio of the query weight to the update weight is very high and almost all the intervals should be marked PUSH and hence RANDOMIZE does poorly. As the number of updates increases, the effect of this anomaly weakens, as we see in Fig. 12.

Effect of Query Costs under Model $\mathcal{M}_3$. Next we considered model $\mathcal{M}_3$, where queries and updates have communication cost which is linear in the size of the transferred data. The algorithms and the heuristics are also modified suitably. For the first two simulations we assume that the size of the transferred data $N(i)$ depends linearly on the number of updates. Specifically we assume that for every 2 updates made, only one new data item is added. The parameters α and β can be interpreted as initiation cost or header cost and data cost respectively. We adopted accepted network values of the ratio between them by choosing $\alpha = 3$ and $\beta = 0.01$. The data values were between $1,000$ and $30,000$. The results are shown in Fig. 13.

Fig. 13(a) shows that at around 18,000 updates, UNIFORM switches from PUSH to PULL. This is because



(a) Varying number of updates. ($\mathcal{M}_3$; 6000 queries.)

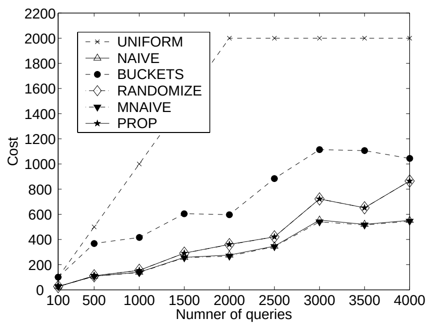


(b) Varying number of queries. ($\mathcal{M}_3$; 10000 updates.)

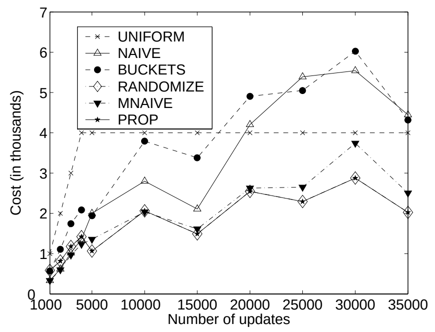
Figure 13: Cost Model $\mathcal{M}_3$. (Query-length distribution: $G(500, 400)$.)

the total query cost grows at half the rate of the update cost. Further, in Fig. 13(b), we clearly see that PROP and RANDOMIZE make poor decisions when the query weight is much larger than the update weight, mostly because of their query-sharing nature. Once they make an error there is a snowballing effect because they remove pulled queries. RANDOMIZE's even worse performance is explained as it was for Fig. 12. Note that the other heuristics perform only marginally worse than DYNPROG.

Multi-attribute Data Gerrymandering. We implemented the multi-attribute versions of the above algorithms (see Section 4.2) and tested them on the synthetic data for two attributes, generated in a manner similar to the single-attribute data, and the number of updates varying from 1,000 to 10,000 when the number of queries was fixed at 4,000. The results are in Fig. 14. When there were 2,000 queries, the cost of the baseline was 2,000, while the cost of MNAIVE and PROP were both less than 400, less by a factor of about 5. In general, the more attributes, the more regions in the partition, and the more are the benefits.



(a) Effect of Number of Queries



(b) Effect of Number of Updates

Figure 14: Two-attribute workloads. Effect of number of queries. ($\mathcal{M}_1$; 2000 updates.)

6.2 Evolving Workloads

There are various possible combinations of the algorithms for evolving workloads (Section 5). We implemented two of the discussed algorithms: (a) DYNNAIVE with the windowing technique,[2] and (b) DENSITY with the windowing technique. We ran these algorithms on a sequence based on the SDSS workload described in Section 6.1.1. The SDSS workload, in fact, has all updates before all queries, and so can be solved efficiently even by our algorithms for static workloads. To create a workload that actually evolves dynamically, we took a random permutation of the queries and updates. For DYNNAIVE we chose $\alpha = 0.5$, $\beta = 0.25$, and for DENSITY we chose $\alpha = 2$. Fig. 15 shows the communication cost incurred by the algorithms at various stages of the sequence. It also shows the cost of a "static" DYNPROG that created a labeled partition based on the first 200 events.

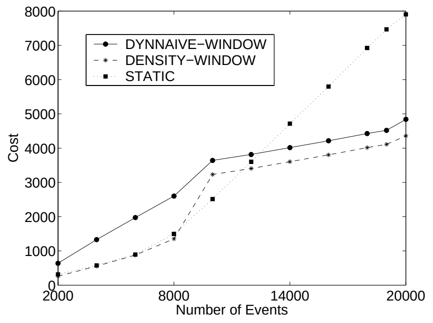


Figure 15: Cost for an evolving SDSS workload ($\mathcal{M}_1$.)

Instead of looking at just the total communication cost incurred we study the performance of the methods for recomputation based on the cost per query or update in the input stream. The method DENSITY is a benchmark of sorts because it keeps the cost density low. The plots suggest how the two algorithms responded to the sequence. Consider the static DYNPROG and DENSITY. To begin with, both had a cost density of 0.2 units per event. Then, at 8,000 events, the workload pattern changed and the cost density for DENSITY rose to 0.94 units per event, and the algorithm responded by recomputing its partition. As a result, its cost density reduced to about 0.1 units per event and remained nearly steady after that. The static DYNPROG, however, was unable to rectify for the changed workload and so its cost density increased to 0.53 units per event. The algorithm DYNNAIVE performed similarly to, but not quite as well as, DENSITY. It too responded to the change in the workload and had a final cost that was about 60% of the static algorithm. Thus in practice DYNNAIVE works reasonably well to keep the cost density low, while being very simple to implement.

7 Conclusions

This paper introduced *data gerrymandering* as a technique for reducing communication cost of range queries on dynamic data in a client-server environment using client-side caching. We formally described this technique, and formulated an optimization problem to find an optimal labeled partition. We developed algorithms for various cases. Our experiments on real-life and synthetic workloads show that this simple but powerful technique significantly reduces communication costs.

[2]Notice that DYNNAIVE resembles the algorithm suggested in [KB96].

References

[AFZ97] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. *ACM SIGMOD Record*, 26(2):183–194, June 1997.

[APTP03] Khalil Amiri, S Park, R Tewari, and S Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *ICDE*, pages 821–831, 2003.

[AWY99] Charu C. Aggarwal, Joel L. Wolf, and Philip S. Yu. Caching on the World Wide Web. *TKDE*, 11(1):95–107, 1999.

[Bar99] Daniel Barbará. Mobile computing and databases: A survey. *IEEE Trans. Knowl. Data Eng.*, 11(1):108–117, 1999.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A multidimensional workload-aware histogram. In *SIGMOD*, pages 211–222, 2001.

[DFJ$^+$96] Shaul Dar, Michael J. Franklin, Bjorn Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[DKP$^+$01] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant J. Shenoy. Adaptive push-pull: Disseminating dynamic web data. In *WWW '01*, pages 265–274, 2001.

[Fra96] Michael J. Franklin. *Client Data Caching: A Foundation for High Performance Object Oriented Database Systems*. Kluwer, 1996.

[GS01] Jim Gray and Alexander Szalay. The World Wide Telescope. Technical Report MSR-TR-2001-77, Microsoft Research, 2001.

[HKU99] Laura M Haas, Donald Kossman, and Ioana Ursu. Loading a cache with query results. In *Proc. of the 25th Intl. Conf. on Very Large Data Bases (VLDB '99)*, pages 351–362, 1999.

[Ji02] Minwen Ji. Affinity-based management of main memory database clusters. *ACM Transaction on Internet Technology*, 2(4):307–339, November 2002.

[KB96] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(1):035–047, 1996.

[KCJH01] Sung Suk Kim, Young Chung, Soon Young Jung, and Chong-Sun Hwang. Optimistic transaction processing algorithms in pure-push and adaptive broadcast environments. In *Proc. 8th Intl. Conf. on Parallel and Distributed Systems*, pages 289–296, 2001.

[KMS00] Nick Koudas, S Muthukrishnan, and Divesh Srivastava. Optimal histograms for hierarchical range queries. In *PODS*, pages 196–204, 2000.

[OLW01] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. In *SIGMOD*, 2001.

[PIHS96] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305, 1996.

[PvST02] G. Pierre, M. van Steen, and A.S. Tanenbaum. Dynamically selecting optimal distribution strategies for web documents. *IEEE Transactions on Computers*, 51(6):637–651, June 2002.

[sdsa] Sloan digital sky survey. http://www.sdss.org.

[sdsb] Sdss data release 1. http://www.sdss.org/dr1.

[SRB97] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. Adaptive data broadcast in hybrid networks. In *VLDB*, pages 326–335, 1997.

[TYD$^+$04] Niki Trigoni, Yong Yao, Alan J. Demers, Johannes Gehrke, and Rajmohan Rajaraman. Hybrid push-pull query processing for sensor networks. In *GI Jahrestagung*, pages 370–374, 2004.

[Wan99] Jia Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Review*, 29(5):36–46, October 1999.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.