Achieving Communication Efficiency through Push-Pull Partitioning of Semantic Spaces to Disseminate Dynamic Information

Amitabha Bagchi, Amitabh Chaudhary,

Michael T. Goodrich, *Senior Member, IEEE,* Chen Li, *Member, IEEE,*

and Michal Shmueli-Scheuer

Abstract

Many database applications that need to disseminate dynamic information from a server to various clients can suffer from heavy communication costs. Data caching at a client can help mitigate these costs, particularly when individual PUSH-PULL decisions are made for the different semantic regions in the data space. The server is responsible for notifying the client about updates in the PUSH regions. The client needs to contact the server for queries that ask for data in the PULL regions. We call the idea of partitioning the data space into PUSH-PULL regions to minimize communication cost *data gerrymandering*. In this paper we present solutions to technical challenges in adopting this simple but powerful idea. We give a provably optimal-cost dynamic programming algorithm for gerrymandering on a single query attribute. We propose a family of efficient heuristics for gerrymandering on multiple query attributes. We handle the dynamic case in which the workloads of queries and updates evolve over time. We validate our methods through extensive experiments on real and synthetic data sets.

Index Terms

Data communications, dissemination, data gerrymandering.

Amitabha Bagchi is with the Department of Computer Science and Engineering at IIT Delhi, India. Amitabh Chaudhary is with the Department of Computer Science and Engineering at the University of Notre Dame, Indiana 46556. The last three authors are with the Department of Computer Science at the University of California, Irvine, California 92697.

I. Introduction

Many emerging applications need to disseminate dynamic information from a server to various clients, especially due to the fact that an increasing amount of information becomes available on the Web, and it can be disseminated to clients in different ways such as the internet and wireless networks. A typical example is real-time traffic report systems, e.g., the Travel Advisory News Network (TANN) [1] and SIGALERT [2]. In such a system, a server provides real-time traffic information to its subscribers. By using a hand-held device, a subscriber can get the traffic conditions of the highways she is interested in. She can ask queries such as "tell me the locations of the accidents (if any) between the Jeffrey Road and the Santa Ana Boulevard on Highway I-5 North." She can also register a set of queries to the server, such as "send me an alert on each weekday from 5 PM to 6 PM whenever the traffic speed between the Jeffrey Road and the Santa Ana Boulevard on Highway I-5 North is below 40 miles per hour." These systems share the following characteristics. (1) The data provided by the server is dynamically changing. (2) It is critical for the latest information to be disseminated to the subscribers so that they can make timely decisions (e.g., for subscribers who are driving). (3) The network bandwidth could be limited since the subscribers are relying on a wireless network to send queries and get information from the server.

Two communication paradigms are widely used to disseminate information. In the PUSH paradigm, the server sends updates to a subscriber every time there are changes to its data. Since the data stored at a subscriber (client) is always up to date, every query can be answered on the client side without contacting the server. This approach is preferable when the data is relatively static compared to the frequency of the queries. In the PULL paradigm, a subscriber contacts the server for each query, and the server does not need to push any updates to the client. This approach is more communication efficient when updates are relatively more frequent than queries.

In this paper we study the following problem: in these systems, how to combine the PUSH and PULL paradigms to reduce communication costs based on the distributions of the queries of each client and the data updates on the server? To see the motivation, consider the traffic report example. The queries asked by a subscriber could have a repetitive pattern. On each workday, she tends to ask for traffic conditions of the highway segments she needs to take between home and work.. If these queries are very frequently asked, relatively to the data updates (changes of traffic conditions) on these segments, then it is communication efficient for the server and the client to use the PUSH paradigm for the data of these segments. On the other hand, the subscriber can ask queries for highway segments not on her list of "frequently asked queries." It is better to use the PULL paradigm for the client to get data for these infrequently asked segments. As a consequence, we can utilize the different distributions of the queries of a client and data updates on the server to partition the entire region of traffic information, and decide a PUSH and PULL paradigm for each area. We call the idea of partitioning the data space into PUSH-PULL regions to minimize communication cost *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.

We study technical challenges when adopting this simple but powerful idea. First, if we knew the pattern of the expected queries at a client and the data-update distribution on the server, what is the best way to partition the space and make a PUSH-PULL decision for each partition? We first study this optimization problem in the case where each client and the server do not have limitation on how many different regions that can used for the two paradigms. We develop algorithms in Section III for the case where the client queries have conditions on a single attribute, such as a segment on a highway. The second challenge is how to do efficient gerrymandering when there are many clients in the system. As a consequence, the server may not be able to have too many semantic regions for a client due to storage and efficient-dissemination reasons. In addition, we also need efficient algorithms for deciding a PUSH/PULL-labeled partition with each client. We study this problem in Section IV, for both the case of a single attribute on query conditions and the general case of multiple attributes. The third challenge is how to do data gerrymandering when the queries on the client and the data updates on the server can change their distributions. The server needs to adapt to these changes by automatically detecting these changes and dynamically deciding a new labeled partition with a client. We present our solution to this problem in Section V. We also conducted extensive experiments on both synthetic data and real data to evaluate our solutions to these challenges (Section VI).

For multiple-attribute data, it is worth emphasizing that we do not necessarily do gerrymandering on all its attributes because of two reasons. (1) Users may ask queries on a subset of these attributes, and the system returns objects including data of other informational attributes. This is true especially in Web-based applications that provide search forms. Typically these forms include only some of the attributes for users to specify conditions on. In this case, the query-condition attributes are candidate gerrymandering attributes, and other attributes will not be considered in gerrymandering. (2) Even for the candidate attributes, we do not need to choose all of them to do gerrymandering. We can choose the gerrymandering attributes based on the skew in updates and queries, the complexity of computing a labeled partition, and the corresponding benefits in reducing communication costs. In Section IV-B we discuss how to choose gerrymandering attributes, and compute a labeled partition once they have been decided.

A. Related work

Semantic data caching was proposed in [3], in which the client maintains a semantic description of the data in its cache, and decides whether it should contact the server for data not in the cache. The data needed from the server is specified as a remainder query. [3] focused on cache replacement policies, without considering data updates on the server. Our work differs from this previous work since we consider the case where the server data is dynamic, and the client site has enough cache to store data.

Our work is closely related to the predicate-based caching scheme proposed in [4], 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. This problem is not studied thoroughly in the other work. Second, the work in [4] 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 [4] (Section IV-A). We also conduct an extensive experimental study on these algorithms.

Among other related work is caching in distributed database systems [5], [6], [7], [8], [9], [9], [10]. For instance, [8], [9] studied how to cache pages of queries on dynamic data to reduce workload on database and web servers. [11], [9] studied how to use cache tables to reduce workload on a database server. In both cases, notification messages are sent from the server to the cache when there are data changes. [12] proposed that each document at a client should have its own associated strategy for deciding whether to be pushed to a replication site or stay at the server. 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 [13]. Some use data affinity to reduce cache misses [14]. Our work differs from these studies since it mainly focuses on how to partition the semantic space of data and compute PUSH/PULL labels in order to minimize communication cost, which is not studied by these works. Thus our work complements the earlier studies.

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 [15]. They suggest a three-level architecture — web server, 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. [16] 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 [17], [18]. The asymmetry inherent in the capabilities of nodes versus base stations in mobile networks makes this kind of capability particularly important [19]. 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, [20] 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 [21]. The difference is that our work assumes each query should be answered using the latest data. A similar idea of combining PUSH and PULL paradigms is developed in [22] to reduce communication cost to answer queries in sensor networks. A related work is [23], which proposes techniques to disseminate dynamic information to clients by using cooperating repositories. In our work, we assume the server needs to disseminate information to clients without such cooperating repositories.

II. DATA GERRYMANDERING

We use an illustrative example to show the motivation of combining the PUSH/PULL paradigms, and formally describe data gerrymandering. Consider a car dealer with information about cars stored in a relational table with the schema `car(make, model, year, mileage, color, price)`. Its data needs to be disseminated to many online car shops. Each shop receives user queries and contacts the car-dealer source to retrieve related data. The following are example client queries: (1) Find cars of `year` $>$ `1998 & price in (5K,9K)`. (2) Find the average price of cars of `color = 'red' & mileage in (30K,50K)`. (3) Find the number of cars satisfying: `model = 'Camry' & mileage` $<$ `90K & price in (3K,8K)`.

Data gerrymandering can be used between a shop (client) and the data source (server). We use the `price` attribute to illustrate its advantages. Fig. 1 shows a workload of the data updates and queries during a time period for a particular shop. 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 $(2, 14)$.

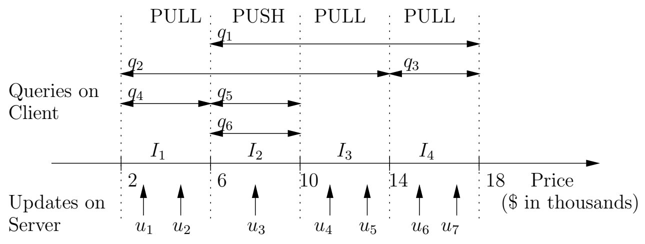


Fig. 1. Queries and updates on cars by price.

Each of these queries needs to be answered using the latest data at the server (the source). Since there could be many queries from the client and data updates on the server, the communication network between them could become a computational bottleneck. If we use the PUSH method for the entire price domain, the server sends 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 contacts the server for each query. Thus the client needs to have 6 interactions with the server for the 6 queries. If our goal is to minimize the number of interactions between them (without considering the size of the transferred data), it is better to use the PULL method due to its fewer interactions.

Interestingly, we can further reduce the number of interactions 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 get the 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. 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 data into different regions and choosing a PUSH/PULL mode for each region, we could reduce the total communication cost. This idea is called data gerrymandering.

A. Formal Definition

Consider applications in which a data source (server) has dynamic data that needs to be disseminated to multiple clients. The server has data about objects, such as information about cars or traffic conditions of different segments on highways). The data is dynamic; new objects can be inserted, existing objects can be deleted, and the values of existing objects can change. Each client issues queries on the data that need to be answered using the *latest* information on the server. Each query specifies conditions on these attributes, and asks for information about objects satisfying the conditions, such as these objects or their aggregated value (e.g., SUM, AVG, MIN). *Data gerrymandering* ("gerrymandering" for short) can be adopted by each client and the server. The client and the server 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.

The Cartesian product of the regions for different attributes forms a partition of the entire semantic space of the data. The client and the server decide such a partition *a-priori*, which consists of disjoint semantic regions. Each semantic region is formed using intervals for these attributes. In addition, both sides decide a PUSH or PULL label for each semantic region. In the example above, we chose `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).

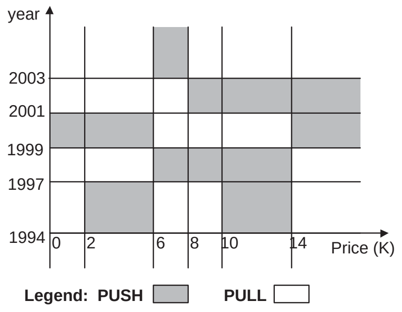


Fig. 2. A labeled partition for the semantic space formed by attributes {price, year}.

After deciding a labeled partition, the client and the server follow a simple protocol. On the server side: it sends the client each update in a PUSH region, and these updates are incorporated into the client cache. These updates are called *pushed updates*. Thus at all times the client has the current data for all the PUSH regions. The updates in PULL regions do not need to be sent to the client. On the client side: For a query whose conditions overlap with PUSH regions only, the client answers it by using its up-to-date data in the cache. Query q_6 in Fig. 1 is an example. On the other hand, for a query that overlaps with PULL regions, the client needs to send to the server a *remainder query* to retrieve the data in these PULL regions, since the client does not have all the latest data in these regions. Formally, a remainder for a query asks for data in the PULL regions overlapping with the conditions of the query. For instance, query q_1 in Fig. 1 is a pulled query, since its condition `6K < price < 18K` overlaps with two PULL intervals I_3 and I_4. To answer this query, in addition to using the cached data in the PUSH interval I_2, the client also needs to send a remainder query to the server to ask for data in the PULL intervals I_3 and I_4.

III. GERRYMANDERING ON A SINGLE ATTRIBUTE

We first study the following problem: given a workload of queries from a specific client and updates on the server that is going to come, how do we find an optimal labeled partition for this client and the server to minimize the communication cost? Such a workload could be obtained by analyzing queries and updates in the past. For instance, in the real-time traffic report example, we can look at the daily queries from a subscriber and the traffic conditions on the highways. This workload can also be obtained from the queries and updates in a time window, if this workload will repeat in the future. We assume that once the labeled partition is decided by the client and the server, it does not change as the workload comes in. We focus on the case of gerrymandering a single attribute.

A. Optimization Problem

Consider a numeric attribute which we want to gerrymander between a specific client and the server. The attribute has a totally ordered domain between a lower bound (possibly $-\infty$) and an upper bound (possibly $+\infty$). Each query from the client has a *range condition* for this attribute, and asks for information about objects satisfying the range condition. We are given a coming workload, which is a sequence of client queries and server 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.

A *communication cost model* $\mathcal{M}$ does the following. Let $\mathcal{A}$ 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 $\mathsf{cost}(u_i)$) when the server sends this update to the client immediately after u_i takes place. 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 $\mathsf{cost}(q_j)$) when the client sends a remainder query to the server to get necessary data for the overlapping PULL regions to answer q_j. The communication cost of a query not intersecting PULL intervals is 0. 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 server does not need to send data in these intervals. Let the set of updates falling in intervals labeled PULL under $\mathcal{A}$

be denoted by $U_{\mathcal{A}}$ and the set of queries overlapping intervals labeled PULL be denoted by $Q_{\mathcal{A}}$. Then the overall cost of workload W under labeling $\mathcal{A}$ is: $\mathsf{cost}(W_{\mathcal{A}}) = \mathsf{cost}(U_{\mathcal{A}}) + \mathsf{cost}(Q_{\mathcal{A}})$, in which $\mathsf{cost}(U_{\mathcal{A}}) = \sum_{u_i \in U_{\mathcal{A}}} \mathsf{cost}(u_i)$, and $\mathsf{cost}(Q_{\mathcal{A}}) = \sum_{q_i \in Q_{\mathcal{A}}} \mathsf{cost}(q_i)$.

Given a workload W of queries and updates and a communication cost model $\mathcal{M}$, our goal is to find a labeled partition $\mathcal{A}$ for the gerrymandering attribute to minimize the communication cost: $\mathsf{cost}(W_{\mathcal{A}})$. Notice that the labeled partition between the client and the server does not change during the expected workload.

B. Model $\mathcal{M}_1$: Unit Cost

We first study the optimization problem for a simple cost model $\mathcal{M}_1$, in which each communication between the client and the server has a unit 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, such as the wireless network in the real-time traffic-report example. This cost model is also relevant in high-bandwidth setting where latency is the main issue. Under this model, the order in which the query/update events happen in the workload do not affect the cost, and we can treat a workload as a set of queries and updates.

We develop a dynamic-programming algorithm, denoted by "DYNPROG," for finding an optimal labeled partition. We begin the description of DYNPROG with a proposition that limits the intervals that need to be considered in order to find an optimal labeled partition.

Proposition 3.1: Given a workload W with a set of updates $\mathbb{U}$ and a set of queries $\mathbb{Q}$, let $p_1 < p_2 < \ldots < p_n$ be the starting and ending points of the ranges in the conditions of $\mathbb{Q}$. (p_1 could be $-\infty$, and p_n could be $+\infty$.) Under the cost model $\mathcal{M}_1$, there is an optimal labeled partition $\mathcal{A}$ whose intervals are either of the form (p_i, p_{i+1}), or the form $[p_i, p_i]$. $\square$

A proof of the proposition is in the Appendix. Based 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.

Notation: An instance of the data gerrymandering problem consists of a workload W with a set $\mathbb{Q}$ of range queries and a set $\mathbb{U}$ of point updates. The problem instance itself is denoted by $\Pi(\mathbb{U}, \mathbb{Q})$. Let the intervals of interest corresponding to $\mathbb{Q}$ be $I_1, \ldots, I_n$ in order. For convenience, we sometimes refer to I_j as simply interval j. A *labeling* or a *labeled partitioning* $\mathcal{A}$ of these intervals assigns each interval to PUSH or PULL and is denoted by a sequence of ordered pairs (I_i, S_i), where $S_i \in \{\text{PUSH}, \text{PULL}\}$, for $1 \leq i \leq n$. If I_j is assigned PUSH, we often denote it by $I_j \in$ PUSH. Finally, let the minimum cost of a labeling for Π be denoted by f.

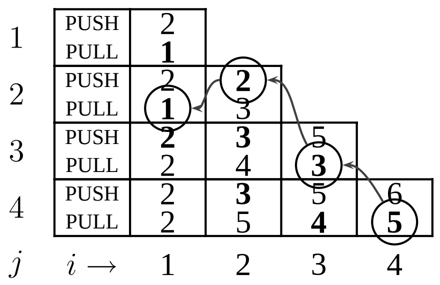
We elaborate the basic idea using the car workload in Fig. 1. If we consider interval I_1 in isolation, we could choose either PUSH or PULL for it, since the number of updates in I_1 is 2, and the number of queries overlapping I_1 is also 2. Of the 2 overlapping queries, q_4 lies entirely within I_1, while q_2 also overlaps with I_2 and I_3. Instead of considering just I_1 in isolation, suppose we had earlier decided to assign PULL to I_2, then the (PULL) cost for q_2 is paid *irrespective* of the choice made for I_1. In that case we would assign PULL to I_1 as such a choice incurs an additional cost of 1 (for the query q_4) whereas a PUSH choice incurs an additional cost of 2. There are further two important observations here: (1) if we decide to assign PULL to I_2, then I_1 would be assigned PULL irrespective of the assignments to the remaining intervals I_3 and I_4; and (2) if instead we decide to assign PUSH to I_2 but PULL to I_3 then by a similar argument (remember q_2 overlaps I_3) I_1 would be assigned PULL irrespective of the assignment to I_4. This example shows that the additional cost for a query in an interval depends on the labels for other later intervals. In general, if a query in the current interval overlaps another interval already assigned 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.

Subproblems and Recurrence Function: To capture the intuition above, in our dynamic programming algorithm we create subproblems from the given workload W and the sets of updates $\mathbb{U}$ and queries $\mathbb{Q}$. Let $\mathbb{U}(i)$ be the subset of updates in intervals I_l, where $1 \leq l \leq i$, and let $\mathbb{Q}(j)$ be the subset of queries that do not intersect any interval I_l, where $(j+1) \leq l \leq n$. For $i \leq j$, the workload consisting of just the updates in $\mathbb{U}(i)$ and queries in $\mathbb{Q}(j)$ is denoted by $W(i,j)$, and the corresponding problem instance is denoted by $\Pi(i,j) = \Pi(\mathbb{U}(i), \mathbb{Q}(j))$. Let the minimum cost of a labeling for $\Pi(i,j)$ be denoted by $f(i,j)$. We shall only care for problems $\Pi(i,j)$ where $0 \leq i \leq j$. Solutions to these subproblems will be used to solve $\Pi(n,n)$ which is the same as our original problem $\Pi(\mathbb{U}, \mathbb{Q})$.

Recall the discussion for the car example; intuitively, $f(i,j)$ is the minimum communication cost for the queries and updates in intervals I_1 to I_i, assuming that I_{j+1} is the next PULL interval after I_i, and all the intervening intervals $I_{i+1}, \ldots, I_j$ are PUSH. DYNPROG is based on a recurrence for $f(i,j)$. There are two cases for the subproblem $\Pi(i,j)$: (1) interval I_i is PUSH, and (2) I_i is PULL. Let $U(i)$ be the number of updates in I_i and $Q(i,j)$ be the number of queries overlapping interval I_i but *not* overlapping interval I_{j+1}. (Notice that $U(i)$ is usually much less than $|\mathbb{U}(i)|$ and $Q(i,j)$ much less than $\mathbb{Q}(j)$.) The recurrence for $f(i,j)$ is given below. In the Appendix we present a formal proof of the correctness of the recurrence.

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

The base case when $i = 0$ is obvious. For $i > 0$, consider the case where I_i is PUSH. This implies that the next closest PULL interval to I_{i-1} is I_{j+1}. Thus, the minimum cost for subproblem $\Pi(i,j)$ in this case can be obtained by taking the adding the minimum cost of the subproblem $\Pi(i-1,j)$ (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 PULL. This implies that the next closest PULL interval to I_{i-1} is I_i. $Q(i,j)$ includes the costs for all queries overlapping interval I_i but not overlapping I_{j+1}. The queries remaining in $\mathbb{Q}(j)$ not accounted for by $Q(i,j)$ are those that only overlap intervals from I_1 to I_{i-1}. Thus, in this case, $\Pi(i,j)$ can be solved by solving the subproblem $\Pi(i-1,i-1)$, and adding $Q(i,j)$ to its cost. 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**

Fig. 3. Computing an optimal solution for the example in Fig. 1.

We use Fig. 3 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: the "PUSH" value (resp. the "PULL" value) corresponds to the minimum cost if interval I_i is in PUSH (resp. PULL) for the subproblem $\Pi(i,j)$. The minimum value (in bold) of the two is the $f(i,j)$ value. Consider, for instance, 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 overlaps interval I_1 and does not overlap 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 $\Pi(n,n)$, 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 minimum $f(n,n)$ value. In particular, if the minimum value for $f(n,n)$ is when it is in the PUSH mode, in the labeling I_n is assigned PUSH and we next look at the entry for $f(n-1,n)$. Again, depending on in which mode has the minimum value, we assign the mode for interval I_{n-1}. 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 next 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. 3 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 II. The optimal cost $f(4,4)$ is 5.

In the Appendix we discuss how to compute the $Q(i,j)$ values efficiently, and prove the following complexity result.

Theorem 3.1: Algorithm DYNPROG runs in $O(nl+m)$ time and takes $O(nl)$ space, where m is the total number of updates and queries, n is the number of intervals of interest, and l is the maximum number of intervals covered by a single query. □

The proof of correctness for DYNPROG (see the Appendix) does not depend on the particular set of intervals $I_1,\ldots,I_n$ used, e.g., they could be different from the intervals of interest of $\mathbb{Q}$ in Proposition 3.1. We formalize this in the theorem below. It is useful when the set of intervals used is much smaller than those in Proposition 3.1 to control the number of the partitions.

Theorem 3.2: Given a workload W with a set of updates $\mathbb{U}$ and a set of queries $\mathbb{Q}$, and a partition of the range (in the conditions of $\mathbb{Q}$) into intervals $I_1,\ldots,I_n$, DYNPROG generates a labeling with the minimum communication cost for the given set of intervals. □

Using intervals of interest of $\mathbb{Q}$ as the intervals for DYNPROG ensures that the labeling generated has the minimum cost irrespective of the set of intervals used.

C. Model $\mathcal{M}_2$: Constant Cost

Cost model $\mathcal{M}_2$ generalizes model $\mathcal{M}_1$ by allowing each event (update or query) to have a constant cost. In cost model $\mathcal{M}_2$, each communication between client and server has a constant cost. Different events can have different costs. This model is applicable in the cases where the costs associated with queries and updates are not necessarily the same. The sources of this cost asymmetry can be various. One typical reason is that the clients and the server have quite different computing resources. Since the server needs to serve many clients, the overhead of answering a query from a client might be different from that of pushing an update to a client, due to complications such as concurrency control. This asymmetry is true, for instance, in mobile networks [19], where the mobile agents have limited power compared to their base station.

Under this model $\mathcal{M}_2$, given a workload W, let $\text{cost}(u_i)$ be the cost of an update u_i in W under this model, and $\text{cost}(q_j)$ be the cost of pulling a query q_j in W. Note that Proposition 3.1 holds for $\mathcal{M}_2$ as well. 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}. This modified algorithm can find an optimal solution under this cost model.

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

Under this cost model, each communication is linear in the size of its transferred data. That is, the cost of transferring data of size s is $\alpha + \beta \times s$, where α and β could be different between pushed updates and pulled queries. Intuitively, this model considers two important types of costs for each event. The α captures the the setup cost such as the initial transfer delay, which could be affected by the overhead on the server. As the number of clients served by the server increases, the server may take more time to process each interaction with a client due to its limited computing resources. The $\beta \times s$ models the costs linear to the size of data transferred. The β value depends on the network bandwidth, and possibly the workload on the server as well. Specifically, each pushed update sends the updated data to the client. For point updates, the updated data is for a single object, but the size of the data transferred can vary from object to object and from update to update. The cost of pushing an update u with parameters α_u, β_u, and size of updated data s_u is $\textsf{cost}(u) = \alpha_u + \beta_u \times s_u$, a constant irrespective of other decisions.

For each pulled query, its cost depends on the PUSH/PULL labels for other intervals. 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 remainder query does not need 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. Let the parameters for a query q be α_q and β_q. Further, let $\mathcal{I}_{\text{PULL}}(q)$ be the intervals overlapping q and assigned PULL by a labeling $\mathcal{A}$. Given an interval I_i let s_{qi} be the size of data transferred by q from I_i. The cost of pulling q under $\mathcal{A}$, $\textsf{cost}(q)$, is $\alpha_q + \sum_{I_i \in \mathcal{I}_{\text{PULL}}(q)} \beta_q \times s_{qi}$. In summary, the cost of a pulled query is no longer a constant, since it depends on the PUSH/PULL labeling.

Gerrymandering under this model is more challenging. In particular, we can show that Proposition 3.1 does not hold true for $\mathcal{M}_3$. To solve this problem, we propose a simple pre-processing step. If first applied to a workload under $\mathcal{M}_3$, it allows us to apply Proposition 3.1 to the resulting modified workload. Consider an object o. Let $\mathbb{U}_o$ be the set of updates on o, and $\mathbb{Q}_o$ be the set of queries that have o within their ranges. Given a query $q \in \mathbb{Q}_o$, let s_{qo} be the size of data from o transferred when q is pushed. If the condition $\sum_{q \in \mathbb{Q}_o} \beta_q \times s_{qo} \geq \sum_{u \in \mathbb{U}_o} \textsf{cost}(u)$ is true, then the interval containing only o should be labeled PUSH, irrespective of other intervals. This is because, intuitively, just the size-dependent data-transfer cost in pulling the (interval containing) object o is more than the cost of pushing the updates on it. Thus pulling the interval containing just the object is always more expensive than pushing it, irrespective of other decisions. In the pre-processing step, we mark all such intervals that satisfy the above condition as PUSH, and then "remove" them from the original workload. By removal, we mean that we ignore updates in such intervals and assume that the size of the data in the intervals is 0. In particular, note that the condition is true for all objects that receive no updates. Let the modified workload be W'.

Proposition 3.2: Proposition 3.1 holds true for W' under cost model $\mathcal{M}_3$. □

The proof is in the Appendix. Now we describe how to modify DYNPROG to find the least cost, assuming we are given a set of intervals for a workload W under $\mathcal{M}_3$. Note that if the workload is W' constructed according to the preprocessing step described above, and the intervals of interest formed according to Proposition 3.1, then we are certain that DYNPROG finds the optimal cost. In general, for any arbitrary set of given intervals, DYNPROG finds the least cost when gerrymandering is constrained to use those given intervals only. In the recurrence function for DYNPROG, 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 becomes

$$\sum_{q \in \mathcal{Q}(i,j)} \alpha_q + \sum_{q \in \mathcal{Q}(i)} \beta_q \times s_{qi} \quad + f(i-1, i-1),$$

in which $\mathcal{Q}(i)$ is the set of queries overlapping I_i, $\mathcal{Q}(i,j)$ is the set of queries overlapping I_i but not I_{j+1}, α_q and β_q are the $\mathcal{M}_3$ cost parameters for q, and s_{qi} is the data transferred by q from I_i. 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 $\mathcal{Q}(i,j)$ intersecting I_i but *not* intersecting interval I_{j+1}, i.e., $\sum_{q \in \mathcal{Q}(i,j)} \alpha_q$. 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 whatever amount of data they do from this interval, corresponding to the "$\beta \times s$" portion in the cost model. This total cost is computed as $\sum_{q \in \mathcal{Q}(i)} \beta_q \times s_{qi}$, where $\mathcal{Q}(i)$ is the set of queries intersecting this interval I_i.

IV. EFFICIENT GERRYMANDERING

In the previous section we studied how to gerrymander a single attribute, assuming that the specific client and the server can partition the entire domain to as many intervals as needed. In this section we relax these assumptions. We consider the cases where a client and the server need to find a labeled partition *efficiently* in both time and space. This requirement is especially critical due to the following reasons. (1) The server could be serving many clients, thus cannot allocate too much space to manage a labeled partition with each client. In addition, it cannot spend too much time running an algorithm for finding a labeled partition for each client. (2) When the workload of queries from a client and the updates on the server keeps changing (Section V), the client and the server need to recompute a labeled partition periodically. In this section, we develop efficient heuristics to find a good labeled partition. For simplicity we focus on the cost model $\mathcal{M}_1$, and the ideas are applicable to other cost models.

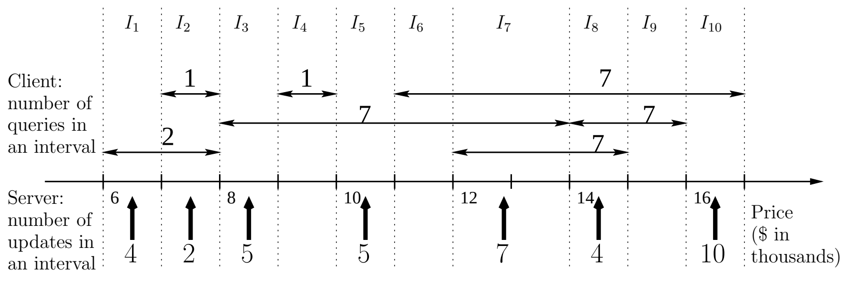
A. Efficient Gerrymandering on a Single Attribute

1) Efficient Partitioning: Given a coming 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. The number of intervals affects not only the space needed by the client and the server to store the labeled partition, but also the time of computing the PUSH/PULL labels.

One approach is to use those intervals of interest formed by the starting points and ending points of the queries in the workload, as defined in Section III-B. This approach might introduce too many intervals when there are many condition ranges from the queries. An alternative approach is to divide the entire domain of the attribute into B intervals of equal width, called "buckets," where B can be decided based on the storage and time need between this client and the server. This approach has the flavor of equi-width histograms [24], which split the range into uniformly-sized 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 [24] 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.

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 labeled partition efficiently. We will use the workload in Fig. 4 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 [4]. 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



I_1 I_2 I_3 I_4 I_5 I_6 I_7 I_8 I_9 I_{10}
Client: number of queries in an interval
1
1
7
7
7
2
7
Server: number of updates in an interval
6 8 10 12 14 16
4 2 5 5 7 4 10
Price ($ in thousands)

Fig. 4. A workload for one range attribute.

the interval (i.e., $U(j)$), we label I_j as PUSH; otherwise, we label it as PULL. In Section VI 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.

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 VI 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.

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. That is, $w(j) = \sum_{q \text{ overlaps } I_x} \frac{1}{l(q)}$, where we denote the number of intervals intersected by a query q as $l(q)$. 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. This method is seen to be much more communication efficient than the others when the number of updates is high (cf. Section VI).

Table I shows the results of running these heuristics and the DYNPROG algorithm on the workload in Fig. 4. The UNIFORM algorithm decides a PULL label for the entire region. Among the 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.

Heuristics	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	$Cost_U$	$Cost_Q$	Total
0. UNIFORM	L	L	L	L	L	L	L	L	L	L	0	32	32
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 I

LABELED PARTITIONS BY DIFFERENT HEURISTICS FOR THE WORKLOAD IN FIG. 4 ("H" FOR "PUSH" AND "L" FOR "PULL"). THE UNIFORM ALGORITHM DECIDES A LABEL FOR THE ENTIRE REGION. THE NEXT 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.

B. Gerrymandering on Multiple Attributes

As mentioned in Section I, for multiple-attribute data, we do not need to gerrymander all its attributes, mainly because users could ask queries on a subset of these attributes, and we have the freedom to choose a subset of gerrymandering attributes. Now we discuss how to choose gerrymandering attributes, and compute a labeled partition for multiple attributes.

Choosing Gerrymandering Attributes: Given a set of candidate attributes, which could be a subset of all the data attributes, we could choose attributes to do gerrymandering as follows. Consider a workload that is expected to come. For each attribute, we run one of the algorithms to find a labeled partition for the given coming 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 algorithm 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 (1) we use all the candidate attributes for gerrymandering, (2) the communication cost does not decrease much, or (3) the computation becomes too expensive.

Computing a Labeled Partition for 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 coming workload to reduce communication cost. 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. This approach produces too many regions, and the client and the server have a limited number of regions to consider. In this case, we can consider a partition formed by equi-width buckets for each attributes, while we can decide the number of buckets for each attribute based on the distribution of queries and updates on that attribute. In principle, many techniques on building multidimensional histograms (e.g., [25]) can be adopted to find a partition.

After deciding a partition, it is straightforward to extend those heuristics to decide PUSH/PULL labels for the semantic regions in the 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.

V. GERRYMANDERING FOR DYNAMICALLY EVOLVING WORKLOADS

There are many situations where the techniques developed so far will serve us adequately for the case where distributions of queries and updates are repetitively changing. 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 a similar workload is expected to be come. For instance, in the 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 distributions 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 distribution of one week to decide a labeled partition, and use it for every week; and correspondingly use the distribution from one weekend as a template for all the weekends in the future.

In this section we study how to do adaptive data gerrymandering where if it is hard to find a repetitive workload. 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 IV-A. 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. To deal with evolving distributions, a client and the server can keep the queries and updates in a time window, and use them to detect substantial changes. These events can also be used as a distribution for the future. Two questions need to be answered. (1) When can we say that the distribution has changed significantly enough to warrant recomputing a labeled partition? (2) How do we efficiently compute a new labeled partition? We study these problems in this sections. To be specific in our discussion, we use the dynamic programming DYNPROG for the single-attribute case as an example, and the ideas are generally applicable to other heuristics in the multiple-attribute case as well.

A. Detecting Changes in Workload Patterns

We first develop criteria for determining when the pattern of the distribution 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 III, 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. 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).

Let the input stream since the dynamic program was previously run on X be denoted 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 (these two parameters are different from those in cost model $\mathcal{M}_3$): 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 PUSH intervals, and higher values to those of PULL intervals. Using the earlier notation, 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 III-A). Denote it by $C(X, I_X, S_X)$.

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

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. 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.

B. 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 because if a region was earlier marked PULL and is now relabeled PUSH, then all the updates from the beginning (possibly before the window began) that have not been pushed have to be pushed now.

VI. EXPERIMENTS

In this section, we present our experimental results to demonstrate the usefulness of data gerrymandering, and compare the performance of the proposed approaches for different scenarios.

A. Experimental Setting

We have set up two experimental environments with a single client and a single server. In the first environment, we simulated both the client and the server using an emulator on a single machine with four Pentium PIII Xeon 500MHz processors with 512KB cache each and overall memory of 4 GB, and a Linux operating system. In the second environment, the server was a Solaris machine with a 502 MHz UltraSPARC CPU and 768 MB memory, running SunOS Release 5.9 Version. The client was a linux machine with two AMD 250Mhz CPUs and 4 GB memory. All the programs were implemented in C compiled using a GNU C compiler.

For both environments, we generated workloads that included a sequence of queries at the client and a sequence of updates at the server. Each event had an arrival time. We controlled a generated workload by adjusting the parameters including the number of queries, the number of updates, the distribution of query lengths, and those parameters in the various cost models. Given a generated workload, we ran different algorithms to compute a labeled partition. In the first experimental environment, for the labeled partition generated by each algorithm, we can compute the cost of each event (query or update) by using the parameters in a cost model. In the second experimental environment, we measured the clock time of each event. For a pulled query, its elapsed time was measured from the moment the query was issued, until the client received all the results. This time should include the time of sending the query, the time for the server to handle the request,, and the time to transfer back the query results. Similarly, for a pushed update, we measured the time from the moment the update was pushed from the server to the client, until the time the server received the acknowledgment from the client. Since the real network in our computing environment had a very high bandwidth and a very small transfer delay, we needed to simulate the parameters in those cost models. There are different ways to simulate these parameters, and we used the system function `sleep()` to simulate the total elapsed time. For instance, in cost model $\mathcal{M}_3$, for a pulled query that needs to retrieve s objects from the server, we let the server process wait for $\alpha + \beta \times s$ milliseconds before sending the results to the client. Our experimental results (described in Appendix) showed that these two environments give consistent results. We conducted other experiments by using the first environment.

Data: We used both a real data set and a synthetic data set. The real data set included real updates and queries from the logs of the Sloan Digital Sky Survey (SDSS) [26]. The synthetic data set included 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 III. 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.

B. Static Workloads

We implemented the optimal algorithm and the heuristics discussed in Sections III and IV: 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, 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.

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.

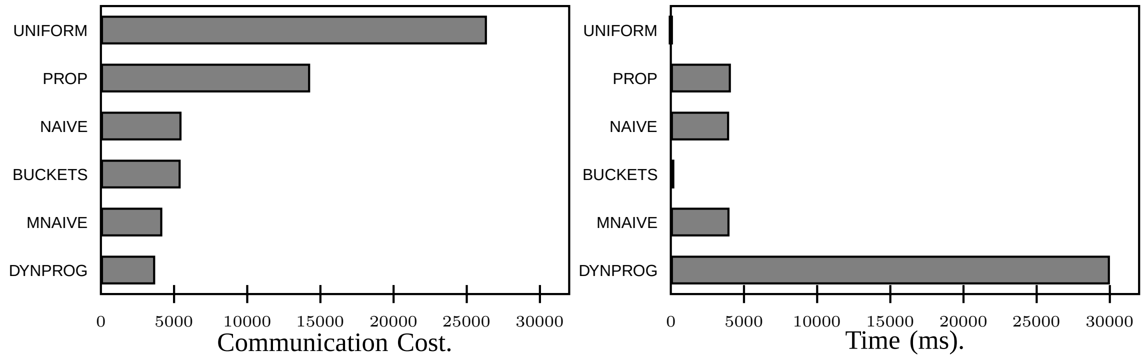


Fig. 5. Communication costs and running times for a real SDSS workload.

We used a selection of 30,000 range queries from the query logs of Data Release 1 of the SDSS [27]. For updates, we used a selection of 50,000 updates in the database between its Early Data Release and the Data Release 1. These queries and updates are effectively random "slices" of the actual queries and updates; they were chosen for us by the SDSS team, without any regard for the performance of gerrymandering on them. In Fig. 5 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.

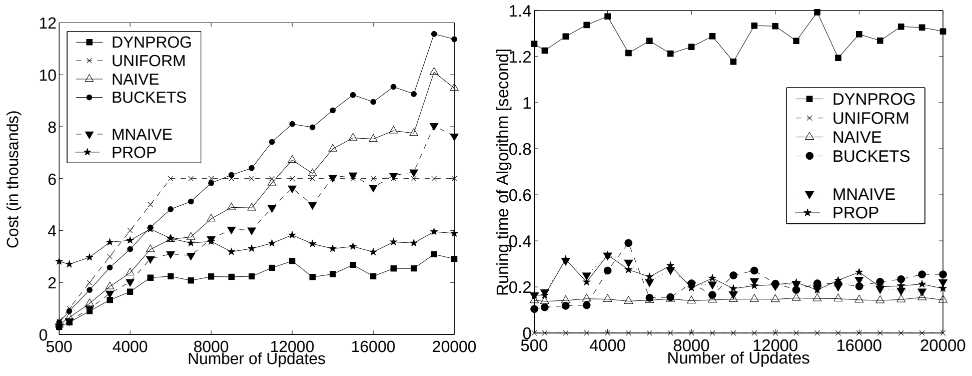
Compared to the cost of the baseline UNIFORM, the cost of the optimal DYNPROG was less by a factor of more than 7. Thus, gerrymandering can indeed save on communication costs. The heuristic MNAIVE had a cost that was more than the optimal by a factor of just 1.13. Its running time was, however, less 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.

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 [28] to generate the Gaussian. We use $G(\mu, \sigma)$ to denote a Gaussian distribution with mean μ and standard deviation σ.

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 IV-B) on two-dimensional data.

Effect of Number of Updates. We first evaluated the algorithms as the number of updates changed. Fig. 6(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 UNIFORM. The reason is that they disregard the possibility of a query overlapping several intervals — as if it occurs entirely within the current interval under consideration. It is equivalent to replacing a



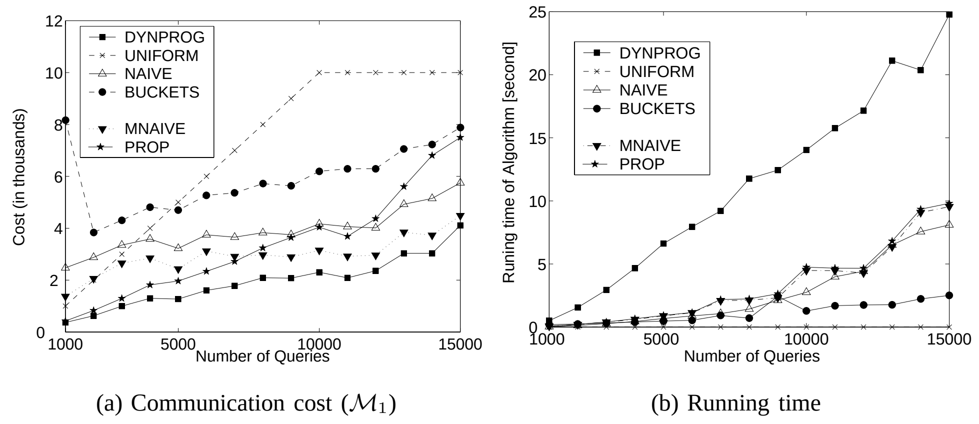
(a) Communication cost ($\mathcal{M}_1$) (b) Running time

Fig. 6. Effect of number of updates. (6000 queries; query-length distribution: $G(500, 400)$.)

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" heuristic PROP does not have a PUSH bias, and performs well for number of updates more than $8,000$. The query-sharing heuristic is 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. 6(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. Fig. 7(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

Fig. 7. Effect of number of queries on communication cost. ($\mathcal{M}_1$; 10,000 updates; query-length distribution: $G(500, 400)$.)

Fig. 7(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 (the effect is roughly a linear increase), 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. 6(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. We also did experiments on the effect of query lengths, and the results are in Appendix.

Effect of Query Costs under Model $\mathcal{M}_2$. Next we considered cost model $\mathcal{M}_2$, and studied the DYNPROG and the heuristics modified suitably for this model. Each update has unit cost. The cost of each query is 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. 8. Consider the communication costs, and compare the results with the earlier ones in Fig. 6 (a). Note that the only difference between the two situations is that now the queries have an average weight of 10. In the earlier results, UNIFORM switched from a PUSH to a PULL when the number of updates increase beyond number of queries, at $6,000$. Here we would expect the switch to take place when the number of updates is about $60,000$, beyond the range shown. Thus, as expected, the new results should be similar to only the “left” half of Fig. 6 (a). In particular, note that the query-sharing heuristic PROP performs much worse than the other heuristics. All the query-grabbing heuristics perform well, with MNAIVE being quite close to DYNPROG.

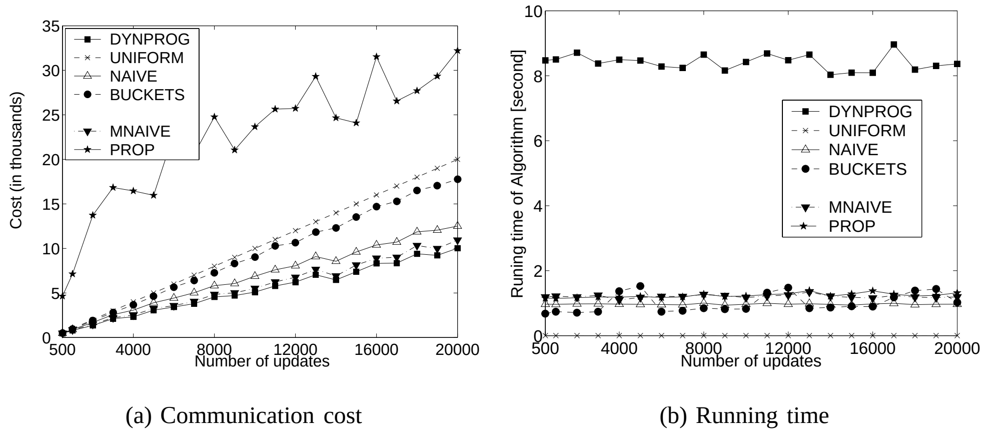
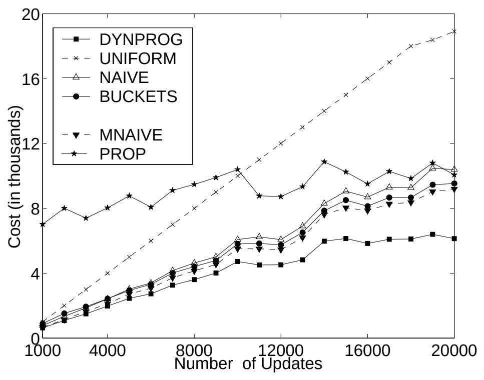


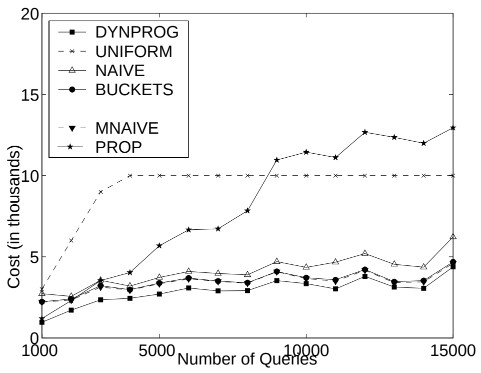
Fig. 8. Effect of query weights on communication cost. ($\mathcal{M}_2$; $6,000$ queries; query-length distribution: $G(500, 400)$.)

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. In these experiments, we directly constructed the intervals of interest as per Proposition 3.1 without any preprocessing as described in Section III-D. As such, the solution of DYNPROG is optimal under the constraint that the intervals of interest be necessarily used. For simplification we assumed that α is 3 and β is 0.01 for all queries. In addition for an interval I_i, each query overlapping it transfers data of size $N(i)$, where $N(i)$ depends linearly on (it is, in fact, half of) the total size of data transferred by updates in I_i. The results are shown in Fig. 9.

Fig. 9(a) shows that at around 18,000 updates, UNIFORM switched from PUSH to PULL, because the total query cost grows at half the rate of the update cost. Compare the results for $\mathcal{M}_3$ with those for $\mathcal{M}_1$: Fig. 9(a) with Fig. 6(a) and Fig. 9(b) with Fig. 7(a). The query-grabbers NAIVE, BUCKETS, and MNAIVE perform better than the query-sharer PROP when the relative number



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

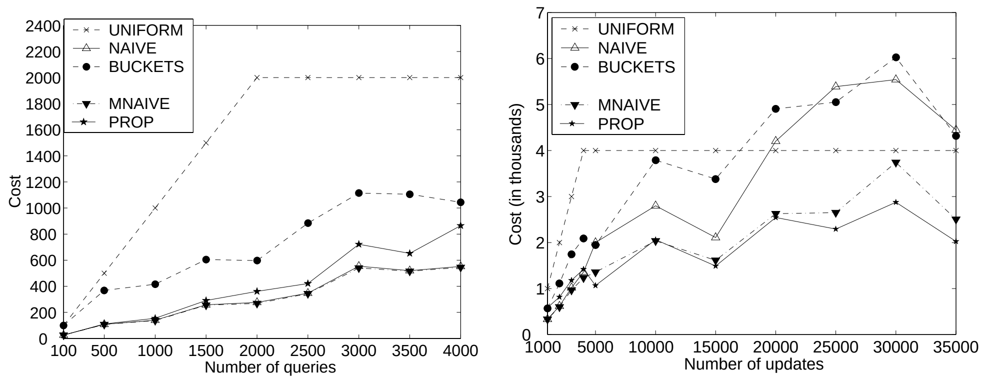


(b) Varying number of queries. ($\mathcal{M}_3$; 10000 updates.)

Fig. 9. Cost Model $\mathcal{M}_3$. (Query-length distribution: $G(500, 400)$.)

of updates is small and vice versa. This is evident in both Fig. 9 (a) and (b). In particular, in Fig. 9, the performance of query-grabbers is worsening, while in Fig. 9 (b) the performance of query-sharer PROP is clearly worsening. In Fig. 9(b), once PROP makes an error there is a snowballing effect because it removes pulled queries. If we compare the results for $\mathcal{M}_3$ with those for $\mathcal{M}_1$, the difference between the better-performing (according to relative number of updates and queries) heuristics and the optimal is less pronounced. (Note that the scales in the figures are different for the two models.) This is to be expected since in $\mathcal{M}_3$ each query has an additional "local" component (β_q) compared to just the "global" component (α_q) in $\mathcal{M}_1$. Since the heuristics tend to make local decisions, more of their decisions are correct in $\mathcal{M}_3$.

Multi-attribute Data Gerrymandering. We implemented the multi-attribute versions of the above algorithms (see Section IV-B) 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. 10. 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

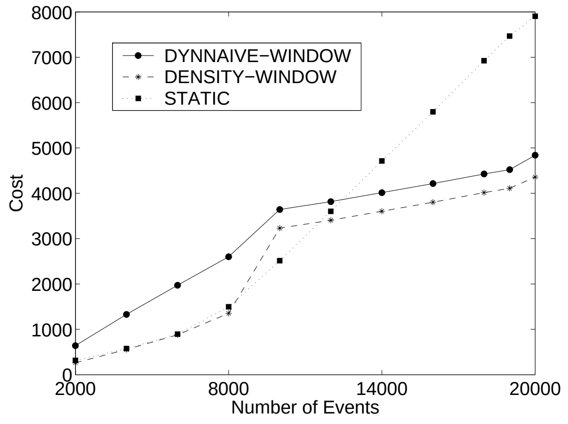
Fig. 10. Two-attribute workloads. Effect of number of queries. ($\mathcal{M}_1$; 2000 updates.)

C. Evolving Workloads

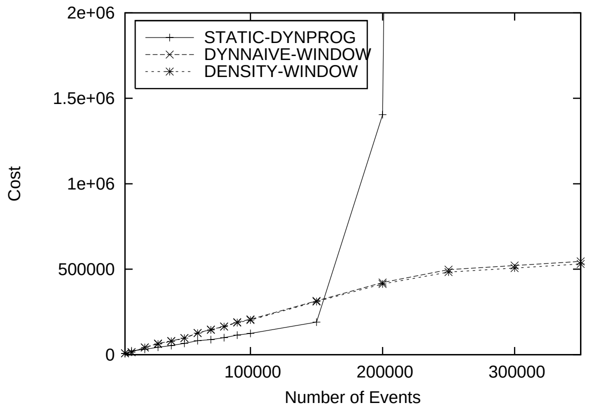
There are various possible combinations of the algorithms for evolving workloads (Section V). We implemented two of the discussed algorithms: (a) DYNNAIVE with the windowing technique,[1] and (b) DENSITY with the windowing technique. We ran these algorithms on two datasets. The first is as described in Section VI-B.1, and the second is similar, except the queries are from Data Release 4, and the updates are from those between Data Release 3 and 4. In addition we used the model $\mathcal{M}_1$ for the first dataset, and $\mathcal{M}_2$ for the second, based on the actual number of tuples returned by the queries on the SDSS database.

The actual SDSS workload, in fact, has all updates before all queries, and so can be solved efficiently even by our algorithms for static workloads. In our first experiment, 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. 11 (a) 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. 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

[1]Notice that DYNNAIVE resembles the algorithm suggested in [4].



(a) Evolving SDSS-based workload under $\mathcal{M}_1$.



(b) Evolving SDSS-based workload under $\mathcal{M}_2$.

Fig. 11. Cost for evolving workloads based on SDSS data.

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.

In our second experiment, we first filtered the SDSS dataset (from Data Release 4), removing queries that did not overlap a minimum threshold number of updates, and removing updates that did not overlap a minimum threshold number of queries. Thus we eliminated the “easy” regions in which even local decisions work well. Then we constructed a workload of 350,000 events by repeatedly choosing SDSS queries and updates in such a manner that the distribution of the queries and updates changes dramatically at 200,000 events. The objective was to stress-test our algorithms for evolving workloads. The costs were modeled using $\mathcal{M}_2$: each update costs 1 unit, and each query costs as many units as the number of tuples returned by the query on the SDSS database. For DYNNAIVE we chose $\alpha = 0.2$, $\beta = 0.25$, and for DENSITY we chose $\alpha = 2$. We plot the results in Fig. 11(b). As expected, for the static algorithm the cost per event jumps to about 614 at 200,000 events from an initial value of about 1. Both DYNNAIVE and DENSITY adapted to the change. They both show remarkably similar behavior, implying that both were triggering recomputations at similar points, and have a final cost per event of about 0.5.

Summary of Experiments: (1) DYNPROG always achieves the least communication cost, but always takes the highest amount of time. (2) The query-grabbing heuristics (NAIVE and MNAIVE) outperform the query-sharing heuristic (PROP) when the number of updates is relatively smaller than the number of queries. The latter is better when the number of queries is relatively smaller than the number of updates. (3) The performance of BUCKETS is variable; there is no guarantee that equi-width partitions result in a good labeling. (4) The performance of MNAIVE is always better than that of NAIVE.

VII. CONCLUSIONS

We introduced *data gerrymandering* as a technique for reducing communication cost of queries on dynamic data in client-server environments using client-side caching. Its idea is to partition the semantic space of a subset of the attributes, and decide a PUSH/PULL label for each region in the space, based on the distributions of queries and updates. We studied the technical challenges when adopting this simple but powerful idea. We developed a dynamic programming algorithm to find an optimal solution for a single gerrymandering attribute, and proposed efficient heuristics to compute labeled partitions. We studied the optimization problem by considering various costs. We discussed how to choose a subset of attributes to do gerrymandering, and how to compute labeled partitions for multiple attributes. We studied how to use gerrymandering for evolving workloads of queries and updates. Our extensive experiments on real-life and synthetic workloads show that this technique can significantly reduce communication costs.

ACKNOWLEDGMENT

Michael T. Goodrich was supported in part by the National Science Foundation under grants CCR-0225642, CCR-0311720, and CCR-0312760. Chen Li and Michal Shmueli-Scheuer were supported in part by the National Science Foundation under CAREER grant IIS-0238586 and IIS-0331707.

REFERENCES

[1] Travel Advisory News Network, "http://traffic.tann.net/."

[2] The SIGALERT System, "http://sigalert.com/."

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

[4] A. M. Keller and J. Basu, "A predicate-based caching scheme for client-server database architectures," *The VLDB Journal*, vol. 5, no. 1, pp. 035–047, 1996.

[5] J. Wang, "A survey of web caching schemes for the Internet," *ACM Computer Communication Review*, vol. 29, no. 5, pp. 36–46, October 1999.

[6] C. C. Aggarwal, J. L. Wolf, and P. S. Yu, "Caching on the World Wide Web," *TKDE*, vol. 11, no. 1, pp. 95–107, 1999.

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

[8] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal, "Enabling dynamic content caching for database-driven web sites." in *SIGMOD Conference*, 2001.

[9] K. S. Candan, D. Agrawal, W.-S. Li, O. Po, and W.-P. Hsiung, "View invalidation for dynamic content caching in multitiered architectures." in *VLDB*, 2002, pp. 562–573.

[10] O. Wolfson and S. Jajodia, "Distributed algorithms for dynamic replication of data," in *PODS*, 1992, pp. 149–163.

[11] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling access to heterogeneous data sources with DISCO," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 5, pp. 808–823, 1998.

[12] G. Pierre, M. van Steen, and A. Tanenbaum, "Dynamically selecting optimal distribution strategies for web documents." *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 637–651, June 2002.

[13] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. J. Shenoy, "Adaptive push-pull: Disseminating dynamic web data," in *WWW '01*, 2001, pp. 265–274.

[14] M. Ji, "Affinity-based management of main memory database clusters," *ACM Transaction on Internet Technology*, vol. 2, no. 4, pp. 307–339, November 2002.

[15] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan, "Dbproxy: A dynamic data cache for web applications," in *ICDE*, 2003, pp. 821–831.

[16] L. M. Haas, D. Kossman, and I. Ursu, "Loading a cache with query results," in *Proc. of the 25th Intl. Conf. on Very Large Data Bases (VLDB '99)*, 1999, pp. 351–362.

[17] S. Acharya, M. Franklin, and S. Zdonik, "Balancing push and pull for data broadcast," *ACM SIGMOD Record*, vol. 26, no. 2, pp. 183–194, June 1997.

[18] S. S. Kim, Y. Chung, S. Y. Jung, and C.-S. Hwang, "Optimistic transaction processing algorithms in pure-push and adaptive broadcast environments," in *Proc. 8th Intl. Conf. on Parallel and Distributed Systems*, 2001, pp. 289–296.

[19] D. Barbará, "Mobile computing and databases: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 11, no. 1, pp. 108–117, 1999.

[20] N. Bruno, S. Chaudhuri, and L. Gravano, "STHoles: A multidimensional workload-aware histogram," in *SIGMOD*, 2001, pp. 211–222.

[21] C. Olston, B. T. Loo, and J. Widom, "Adaptive precision setting for cached approximate values," in *SIGMOD*, 2001.

[22] N. Trigoni, Y. Yao, A. J. Demers, J. Gehrke, and R. Rajaraman, "Hybrid push-pull query processing for sensor networks," in *GI Jahrestagung*, 2004, pp. 370–374.

[23] S. Shah, K. Ramamritham, and P. J. Shenoy, “Maintaining coherency of dynamic data in cooperating repositories.” in *VLDB*, 2002, pp. 526–537.

[24] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita, “Improved histograms for selectivity estimation of range predicates,” in *SIGMOD*, 1996, pp. 294–305.

[25] M. Muralikrishna and D. J. DeWitt, “Equi-depth multidimensional histograms,” in *SIGMOD ’88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1988, pp. 28–36.

[26] “Sloan digital sky survey,” http://www.sdss.org.

[27] “Sdss data release 1,” http://www.sdss.org/dr1.

[28] N. Koudas, S. Muthukrishnan, and D. Srivastava, “Optimal histograms for hierarchical range queries,” in *PODS*, 2000, pp. 196–204.

Amitabha Bagchi received his B. Tech. in Computer Science and Engineering from the Indian Institute of Technology, Delhi in 1996 and his PhD from Johns Hopkins University in 2002. He is currently an Assistant Professor at the Indian Institute of Technology, Delhi. His research interests include algorithms, structural properties of networks and data structures.

Amitabh Chaudhary received a B.Tech. from I.I.T., Kharagpur, in 1992, an M.Tech. from I.I.T., Mumbai, in 1996, and a Ph.D. from Johns Hopkins University, Baltimore, in 2002. His research interests include online algorithms for data caching and network routing, and graph theory. He is currently an assistant professor in Computer Science and Engineering at University of Notre Dame.

Michael Goodrich received his PhD from Purdue University in 1987 and is currently a Professor of Computer Science at University of California, Irvine. Dr. Goodrich’s research is directed at the design of high performance algorithms and data structures for solving large-scale problems motivated from information assurance and security, the Internet, information visualization, and geometric computing. He is also interested in computer science education. He is a member of ACM and a senior member of IEEE.

Chen Li is an assistant professor in the Department of Computer Science at the University of California, Irvine. He received his Ph.D. degree in Computer Science from Stanford University in 2001, and his M.S. and B.S. in Computer Science from Tsinghua University, China, in 1996 and 1994, respectively. He received a National Science Foundation CAREER Award in 2003. He is also a Visiting Research Scientist at Google. His research interests are in the fields of database and information systems, including data integration, data exchange, data warehousing, data quality, and data privacy.

Michal Shmueli-Scheuer received her B.Sc (cum laude) and M.Sc degrees in Information System Engineering from the Technion - Israel Institute of Technology in 2000 and 2002, respectively. Currently she is a PhD student in the Department of Computer Science at the University of California, Irvine. Her research interests include data integration and data management in distributed databases and internet-based systems.

APPENDIX

A. *Proof of Proposition 3.1 in Section III-B*

Proof: We need to show, essentially, that it possible to obtain an optimal labeled partition without considering intervals of the form (s, t), (s, p_i), (p_i, t), or $[s, s]$, in which s, t are not endpoints of the ranges in $\mathbb{Q}$. As shown in Fig. 12, consider a labeled partition $\mathcal{A}$ and two consecutive endpoints p_i and p_{i+1} of the range conditions in the queries. The cost of the labeling $\mathcal{A}$ is denoted by $\textsf{cost}(\mathcal{A})$. Suppose $\mathcal{A}$ 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, which effectively removes the intervals with endpoints at s.) Modify $\mathcal{A}$ by merging I_a and I_b into one PULL interval and thus get a new labeled partition $\mathcal{A}'$ which does not have an interval with an endpoint at s. Now we show that under $\mathcal{M}_1$, we always have $\textsf{cost}(\mathcal{A}') \leq \textsf{cost}(\mathcal{A})$. The reason is that although $\mathcal{A}'$ has a region I_a that is now PULL, but was PUSH in $\mathcal{A}$, no query overlapping I_a is PULL in $\mathcal{A}'$ that was not also PULL in $\mathcal{A}$. In fact, it is feasible that $\textsf{cost}(\mathcal{A}')$ is actually less than $\textsf{cost}(\mathcal{A})$ since updates in I_a that were PUSH in $\mathcal{A}$ are not so in $\mathcal{A}'$. We repeat this merging process until we get a new labeled partition that is as specified in the proposition. ■

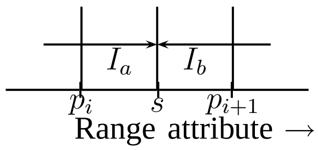


Fig. 12. Intervals of interest.

B. *Correctness proof of* DYNPROG *in Section III-A*

We give a formal proof of correctness for the DYNPROG algorithm. 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., $\Pi(\mathbb{U}, \mathbb{Q}; I_i \in \textsc{push})$ is the problem in which the solution is the minimum cost labeling that assigns PUSH to I_i. Define analogously, $\Pi(i, j; C)$ and $f(i, j; C)$.

Recall that a labeling $\mathcal{A}$ for $\Pi(\mathbb{U}, \mathbb{Q})$ is a sequence of ordered pairs (I_i, S_i), where $S_i \in \{\textsc{push}, \textsc{pull}\}$, for $1 \leq i \leq n$. The subproblem $\Pi(i, j)$, $0 < i \leq j$, doesn't have any updates or queries in intervals $I_{j+1}, \ldots, I_n$, and thus even a sequence of just j ordered pairs is a sufficient labeling. In fact, a sequence of just i ordered pairs is sufficient as well: in $\Pi(i, j)$ intervals $I_{i+1}, \ldots, I_j$ do not have any updates and so, without loss of generality, these intervals can all be labeled PUSH. Thus we use an ordered sequence of i pairs to denote a labeling for $\Pi(i, j)$ with the understanding that the labels for $I_{i+1}, \ldots, I_j$ are all PUSH unless otherwise specified. We use the expression $\mathcal{A} \cup (I_i, \textsc{pull})$ to denote a labeling in which all intervals have their labels as specified by $\mathcal{A}$, except for I_i which is PULL. $\text{cost}(\mathcal{A})$ denotes the cost of $\mathcal{A}$.

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 1.1 and Lemma 1.2.

Lemma 1.1: $f(i, j) = \min\{f(i, j; I_i \in \textsc{push}), f(i, j; I_i \in \textsc{pull})\}$. □

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

Lemma 1.2: For $0 < i \leq j$, $f(i, j; I_i \in \textsc{push}) = U(i) + f(i-1, j)$ and $f(i, j; I_i \in \textsc{pull}) = Q(i, j) + f(i-1, i-1)$. □

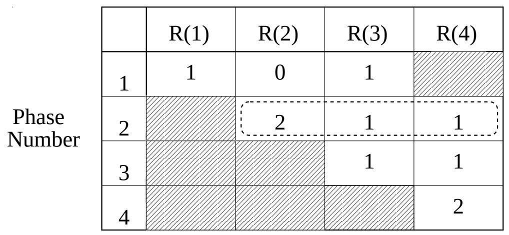
Proof: Consider the problem $\Pi(i, j; I_i \in \textsc{push})$. Let $\mathcal{A}'$ be the optimal partition for $\Pi(i-1, j)$. Now, $\mathcal{A} = \mathcal{A}' \cup (I_i, \textsc{push})$ is a partition for $\Pi(i, j; I_i \in \textsc{push})$. We claim that the $\text{cost}(\mathcal{A}) = \text{cost}(\mathcal{A}') + U(i)$. 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 $\textsf{cost}(\mathcal{A}')$, 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 $\textsf{cost}(\mathcal{A}')$. 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 labeling $\mathcal{B}$ with strictly lower cost for $\Pi(i, j; I_i \in \text{PUSH})$. Clearly, $\mathcal{B}$ can be decomposed into $\mathcal{B}' \cup (I_i, \text{PUSH})$, where $\mathcal{B}'$ is a partition for $\Pi(i-1, j)$. As before, $\textsf{cost}(\mathcal{B}) = \textsf{cost}(\mathcal{B}') + U(i)$. This implies that $\textsf{cost}(\mathcal{B}')$ is strictly less than $\textsf{cost}(\mathcal{A}')$, the optimal cost for $\Pi(i-1, j)$. This is a contradiction.

The proof corresponding to the problem $\Pi(i, j; I_i \in \text{PULL})$ is similar and presented here for completeness. Let $\mathcal{A}'$ be the optimal labeling for the problem $\Pi(i-1, i-1)$. Now, $\mathcal{A} = \mathcal{A}' \cup (I_i, \text{PULL})$ is a labeling for $\Pi(i, j; I_i \in \text{PULL})$. We claim that $\textsf{cost}(\mathcal{A}) = \textsf{cost}(\mathcal{A}') + Q(i, j)$. 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 $\textsf{cost}(\mathcal{A}')$. Now consider pull queries. For every pull query in $\mathcal{A}$ there is either a corresponding pull query in $\mathcal{A}'$ or the query overlaps I_i, but not both (since no query in $\mathbb{Q}(i-1)$ overlaps I_i, and all the intervals $I_{i+1}, ..., I_j$ are marked PUSH in $\mathcal{A}$). In the former case its cost in included in the term $\textsf{cost}(\mathcal{A}')$ 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 labeling for $\Pi(i, j; I_i \in \text{PULL})$, assume for a proof by contradiction that there exists a labeling $\mathcal{B}$ with strictly lower cost. Again, as before, $\mathcal{B}$ can be decomposed into $\mathcal{B}' \cup (I_i, \text{PULL})$, where $\mathcal{B}'$ is a labeling for $\Pi(i-1, i-1)$. It also follows that $\textsf{cost}(\mathcal{B}) = \textsf{cost}(\mathcal{B}') + Q(i, j)$. This implies that $\textsf{cost}(\mathcal{B}')$ is strictly less than $\textsf{cost}(\mathcal{A}')$, the optimal cost for $\Pi(i-1, i-1)$. This is a contradiction. ■

C. *Computing $Q(i, j)$ Values Efficiently in* DYNPROG

We discuss how to compute $Q(i, j)$ values Efficiently in DYNPROG, as described in Section III-A.

We compute the $Q(i, j)$ values in n phases, where n is the number of intervals of interest. Begin sequence of queries sorted first by their start points and then , in case of ties, by their end points. Let l be the maximum number of intervals intersected by any single query. In phase i, we compute the $Q(i, j)$ values for $j = i, \ldots, i + l - 1$. 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

Fig. 13. 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. 13 shows a run of this computation on the example given in Fig. 1.

D. *Complexity analysis of* DYNPROG

As before, n is the number of intervals of interest and l is the maximum number of intervals intersected by any single query. The number of queries in the workload is denoted by m_q, and the number of updates is denoted by m_u. Therefore, the size of the input is $m = m_q + m_u$. A simple geometric argument shows that $(n+1) \leq 2m_q$. (If we assume all queries are distinct we can also say $2m_q \leq n(n-1)$.)

For the space complexity we just need to consider the size of the table for $f(i,j)$ and the space required to compute and store the $Q(i,j)$ values. Both these terms are $O(nl)$. For the time complexity, most tasks, e.g., 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. Thus algorithm DYNPROG runs in $O(nl + m)$ time and takes $O(nl)$ space.

E. Proof of Proposition 3.2 in Section III-D

Proof Sketch of Proposition 3.1 for W' under $\mathcal{M}_3$. Consider the intervals I_a and I_b in Figure 12. Let it be that in optimal labeling $\mathcal{A}$, $I_a \in$ PUSH and $I_b \in$ PULL. Merge I_a and I_b into one PULL region to get $\mathcal{A}'$. $\mathsf{cost}(\mathcal{A}')$ is the same as $\mathsf{cost}(\mathcal{A})$ except that (1) we add the additional size-dependent data-transfer costs for the queries that were originally PULL only for I_b in $\mathcal{A}$ and are now PULL for I_a as well in $\mathcal{A}'$, and (2) we subtract the cost due to the updates in I_a that were PUSH in $\mathcal{A}$ but are not so in $\mathcal{A}'$. The pre-processing step described earlier ensures that in W', the cost subtracted in (2) is more than the cost added in (1). $\square$

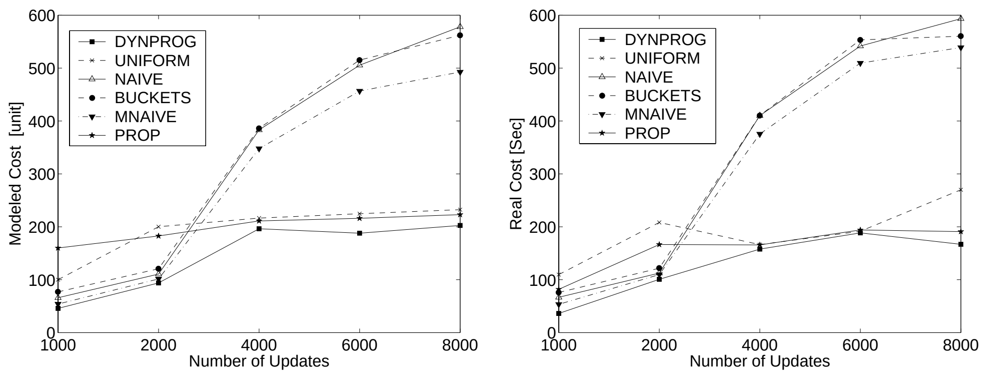
F. Comparison of Two Experimental Environments

To see if the two environments described in Section VI-A give consistent results, we generated a sequence of updates and queries, with their intervals of arrival times following a Poisson process with the mean of 300 and standard deviation of 100. The number of queries was set to 2000; the condition range of the queries was $[1000, 10000]$; the length of queries followed a Gaussian distribution, with the mean of 300 and standard deviation of 200. We used the cost model $\mathcal{M}_3$, and set $\alpha = 100$ and $\beta = 0.1$ in both settings.

We evaluated the algorithms in these two environments. Figures 14(a) and (b) show the results of the two experimental environments. Figure 14(a) is the results of the first experimental environment, and the y-axis is using the normalized cost. Figure 14(b) is the results of the second experimental environment, and the y-axis is using the total elapsed time (in seconds). They show that the results in the two environments are very consistent. Since the second environment needs much more time to run different experiments, in order to run many rounds of experiments to evaluate the effect of different parameters, in the remaining experiments, we used the first experimental environment.

G. Experiments on Effect of Query Lengths

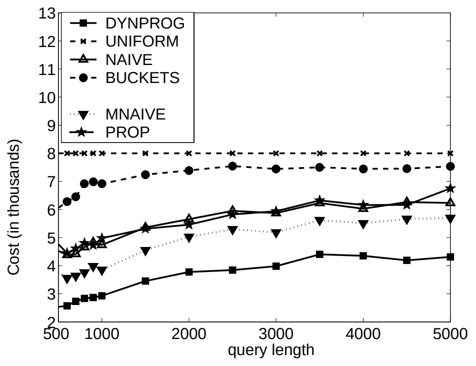
We did a set of experiments to evaluate the effect of query lengths on the performance of different methods. The setting was described in Section VI-B. The range attribute values were between $1,000$ and $30,000$. The number of updates, as well as the number of queries, were fixed at $8,000$. We let the (mean of the) query lengths vary from 500 to $5,000$. The results are shown in Fig. 15. Regarding the communication cost, observe that the relative performance of the



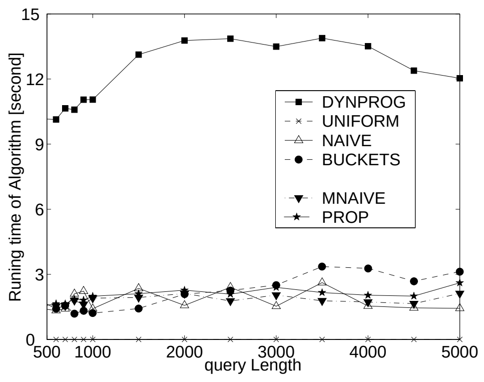
(a) Single-machine emulation environment. (b) Two-machine environment.

Fig. 14. Results of two experimental environments

heuristics does not change with increasing query lengths. Since we did not design any particular heuristic to perform better than any other heuristic when query lengths increase, we consider this result interesting, but not surprising. Further, the cost for each algorithm increases with increasing query lengths; the increase is more pronounced at first, and then seems to saturate. This is as expected. As a query's length increases the cost of pulling it remains the same, but the cost of pushing it increases as it now overlaps more updates. Increasing query lengths, in general, result in longer intervals, containing more updates. As a result, the cost due to an interval labeled PUSH keeps rising — which increases the total cost — until it becomes more economical to label the interval PULL. As more such intervals are labeled PULL the increase in the total cost is less pronounced. We note that MNAIVE performs quite well, and BUCKETS performs quite poorly. As discussed earlier, with increasing query lengths very few buckets are labeled PUSH. DYNPROG outperforms all others in terms of the communication cost but has the longest running time.



(a) Communication cost



(b) Running time

Fig. 15. Effect of query lengths on communication cost. ($\mathcal{M}_1$; $8,000$ updates; $8,000$ queries.)