APPROXIMATE REPLICATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Christopher Alden Remi Olston
June 2003

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Michael Franklin
(UC Berkeley CS Department)

Approved for the University Committee on Graduate Studies.

Abstract

In distributed environments that collect or monitor data, useful data may be spread across multiple distributed nodes, but users or applications may wish to access that data from a central location. A common way to facilitate centralized access to distributed data is to maintain *replicas* of data objects of interest at a central location. When data collections are large or volatile, keeping replicas consistent with remote master copies poses a significant challenge due to the large communication cost incurred. Consequently, in many real-world environments exact replica consistency is not maintained, and some form of inexact, or approximate, replication is typically used instead. *Approximate replication* is often performed by refreshing replicas periodically. Periodic refreshing allows communication cost to be controlled, but it does not always make good use of communication resources: In between refreshes some remote master copies may change significantly, leaving replicas excessively out of date and inaccurate, and meanwhile resources may be wasted refreshing replicas of other master copies that remain nearly unchanged.

This dissertation studies the problem of making better use of communication resources in data replication environments than approaches based on periodic refreshing. In this dissertation, analysis of approximate replication environments is framed in terms of a two-dimensional space with axes denoting system *performance* (a measure of communication resource utilization) and replica *precision* (a measure of the degree of synchronization with remote master copies). There is a fundamental and unavoidable tradeoff between precision and performance: When data changes rapidly, good performance can only be achieved by sacrificing replica precision and, conversely,

obtaining high precision tends to degrade performance. Two natural and complementary methods for working with the precision-performance tradeoff are proposed to achieve efficient communication resource utilization for replica synchronization:

1. Maximize replica precision in the presence of constraints on communication cost.

2. Minimize communication cost in the presence of constraints on replica precision.

Problem definition, analysis, algorithms, and implementation techniques are developed for each method in turn, with the overall goal of creating a comprehensive framework for resource-efficient approximate replication. The effectiveness of each technique is verified using simulations over both synthetic and real-world data. In addition, a test-bed network traffic monitoring system is described, which uses some of the approximate replication techniques developed in this dissertation to track usage patterns and flag potential security hazards.

Acknowledgments

I wish first of all to acknowledge the immeasurably valuable guidance and support I have received from my Ph.D. advisor, Jennifer Widom. The effort she puts into every ounce of the heavy task of advising a Ph.D. student is astounding. She consistently provides top-notch advice on aspects of research and teaching ranging from big ideas to minutiae. Jennifer is a model advisor. I have always been able to count on her to improve the precision and clarity of my writing and speech, and most importantly to help transform my fledgling ideas into crisp and focused research endeavors.

Much of the work presented in this dissertation was done in close collaboration with Jennifer Widom, and portions also reflect collaborations with Jing Jiang, Boon Thau Loo, and Jock Mackinlay. I am grateful to have had the opportunity to work with each of them. My gratitude also extends to Mike Franklin and Hector Garcia-Molina, who have provided helpful feedback on my work over the years and have kindly offered to serve as readers for this dissertation. This work was funded through generous grants from the National Science Foundation and a Chambers Stanford Graduate Fellowship.

I also wish to thank all the members of the Stanford Database Group for countless helpful suggestions and actions. In particular, I would like to thank the following individual Stanford Database Group members for their invaluable assistance: Junghoo Cho, Taher Haveliwala, Andy Kacsmar, Rajeev Motwani, Marianne Siroker, Jeff Ullman, and Calvin Yang. Others who have provided particularly helpful suggestions, feedback, and guidance include Alex Aiken, Russ Altman, David Cheriton, Ed Chi, Michael Chu, David DeWitt, Vuk Ercegovac, Mema Roussopoulos, Nick Roussopoulos, Ion Stoica, Mike Stonebraker, Suresh Venkatasubramanian, and Dapeng Zhu.

Finally, I acknowledge that I would certainly not be in my present position were it not for the thoughtful guidance of Joe Hellerstein, Mybrid Spalding, and Allison Woodruff. I had the pleasure of working closely with Joe, Mybrid, and Allison while I was an undergraduate student at UC Berkeley, and they always treated me as an equal.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In distributed environments that collect or monitor data, useful data may be spread across multiple distributed nodes, but users may wish to access that data from a central location. One of the most common ways to facilitate centralized access to distributed data is to maintain copies of data objects of interest at a central location using an operation called *replication*, as illustrated abstractly in Figure 1.1. In a typical replication environment, a central *data repository* maintains copies, or *replicas* of data objects whose *master copies* are spread across multiple remote and distributed *data sources*. (In general there may be multiple data repositories, but to simplify exposition we focus on a single repository.) Replicas are kept synchronized to some degree with remote master copies using communication links between the central repository and each source. In this way, querying and monitoring of distributed data can be performed indirectly by accessing replicas in the central repository.

While querying and monitoring procedures tend to become simpler and more efficient when reduced to centralized data access tasks, a significant challenge remains: that of performing data replication efficiently and effectively. Ideally, replicas of data objects at the central repository are kept exactly consistent, or synchronized, with the remote master copies at all times (modulo unavoidable communication latencies, of course). However, propagating all master copy updates to remote replicas may be infeasible or prohibitively expensive: data collections may be large or frequently updated, and network or computational resources may be limited or only usable at a

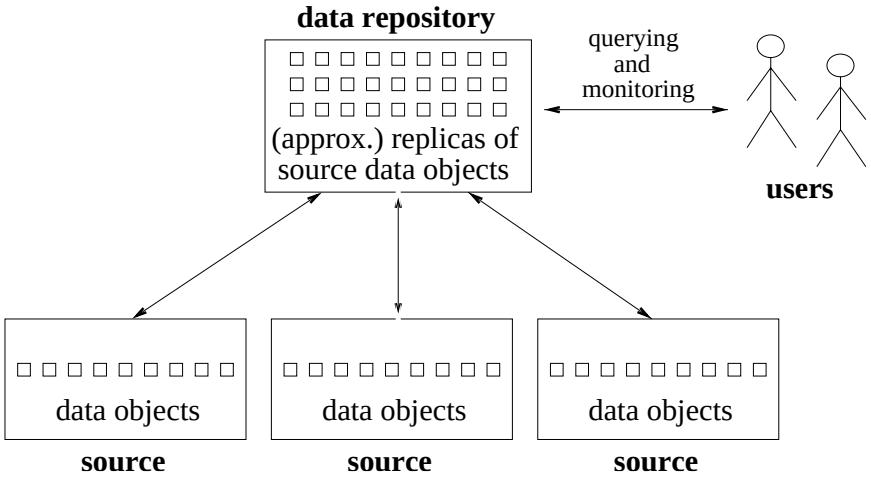


Figure 1.1: Abstract replication architecture.

premium.

Situations where exact replica consistency is infeasible or only feasible at excessive expense can be found in many contexts. As one example, consider sensors that continuously monitor environmental conditions such as sound, wind, vibration, etc. Due to recent advancements, it should soon be possible and relatively cheap to deploy large numbers of battery-powered sensors that communicate via wireless links [36, 59, 91]. Since hundreds of thousands of sensors may be involved, sensor readings may change frequently, and available bandwidth tends to be low in wireless environments, it is not generally possible to propagate every new sensor measurement to a central location for monitoring. Furthermore, even relatively light communication over wireless radio links can consume considerable power, and maximizing battery longevity is a primary goal when deploying small autonomous sensors [75, 80, 91], so communication must be performed sparingly. Similar problems arise in other environments that use wireless or other low-bandwidth links to maintain replica consistency: when volatile data is replicated on portable devices such as PDA's or even recently developed wristwatch devices.

Even in environments that use conventional wired networking, exact replica consistency may still be infeasible due to large quantities of rapidly changing data. For example, in video conferencing applications (*e.g.*, [33]), the viewer screen frame-buffer

can be thought of as containing replicas of video data generated by remote cameras. Since streaming video data can be very large, it often becomes necessary to allow some staleness on parts of the screen. As another example, consider the important problem of monitoring computer networks in real-time to support traffic engineering, online billing, detection of malicious activity, etc. Effective monitoring often requires replicating at a central location a large number of measurements captured at remote and disparate points in the network. Attempting to maintain exact consistency can easily result in excessive burden being placed on the network infrastructure, thus defeating the purpose of monitoring [53], but fortunately network monitoring applications do not usually require exact consistency [112]. As a final example, consider the problem of indexing the World-Wide Web. Keeping an up-to-date Web index requires maintaining information about the latest version of every document. Currently, Web indexers are unable to maintain anything close to exact consistency due to an astronomical number of data sources and data that is constantly changing.

The infeasibility of exact replication in these environments and others is due in large part to the potentially high communication costs incurred. Communication cost tends to be of significant concern in many distributed environments, either because the bandwidth available on the network links is limited (relative to the size and update rate of the data collection), or because network resources can only be used at some premium. This premium for network usage may stem from the fact that increased congestion may cause service quality degradation for all applications that use the network. Alternatively, the premium may be manifest as a monetary cost, either in terms of direct payment to a service provider or as a loss of revenue due to an inability to sell consumed resources to others. As a result of communication resources being a valuable commodity, we have seen that in the applications described above (sensor monitoring, replication on personal wireless devices, video conferencing, network monitoring, and Web indexing), maintaining replicas exactly synchronized with master copies cannot be achieved when data volumes or change rates are high relative to the cost or availability of bandwidth capacity.

1.1 Approximate Replication

In many applications such as the ones described above, exact consistency is not a requirement, and replicas that are not precisely synchronized with their master copies are still useful. For example, approximate readings from meteorological sensors often suffice when performing predictive modeling of weather conditions. In network security applications, "ball-park" estimates of current traffic levels can be used to detect potential denial-of-service attacks. Since exact data is often not a requirement in applications that rely on replication, it is common practice to use inexact replica consistency techniques such as periodic refreshing to conserve communication cost. We use the term *approximate replication* to refer collectively to all replication techniques that do not ensure exact consistency. Most existing approximate replication techniques for single-master environments can be classified into one of two broad categories based on the way they synchronize replicas[1]:

Periodic pushing: Sources propagate changes in master copies to the central data repository periodically, sometimes in large batches.

Periodic pulling: The central data repository accesses remote sources periodically to read master copies of data objects and update local replicas as necessary.

One motivation for performing periodic pushing is that one-way messages can be used in place of more expensive, round-trip ones. Also, sources can control the amount of their local resources devoted to replica synchronization. Periodic pulling, on the other hand, has the advantage that sources are not required to be active participants in the replica synchronization protocol. Instead, they need only respond to data read requests from the central repository, a standard operation in most environments. A principal feature shared by both these approaches is that the cost incurred for consumption of communication resources is bounded and controllable, which is not in general the case with exact synchronization methods. However, periodic pushing

[1]Depending on the environment, it may not be practical or possible for sources to communicate with each other, so we assume that such communication is not allowed and synchronization of replicas is performed directly between each source and the central repository. Studying environments amenable to efficient intersource communication is a topic of future work.

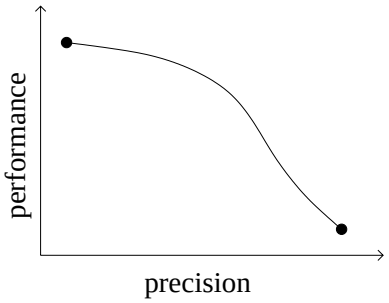


Figure 1.2: Precision-performance tradeoff.

or pulling does not necessarily make good use of communication resources for the following two reasons:

1. Communication resources may be used wastefully while refreshing replicas of data objects whose master copy has undergone little or no change.

2. When the master copy of a data object undergoes a major change that could be propagated to the remote replica relatively cheaply, there may be a significant delay before the remote replica is refreshed to reflect the change.

1.2 Precision-Performance Tradeoff

We study the problem of making better use of communication resources in approximate replication environments than approaches based on periodic pulling or pushing. Our work begins with the observation that a fundamental tradeoff exists between the communication cost incurred while keeping data replicas synchronized and the degree of synchronization achieved. We refer to this characteristic property as the *precision-performance tradeoff*, illustrated in Figure 1.2, where *precision* is a measure of the degree of synchronization between a data object's master copy and a remote replica, and *performance* refers to how sparingly communication resources are used (*i.e.*, the inverse of communication cost). When data changes rapidly, good performance can only be achieved by sacrificing replica precision and, conversely, obtaining high precision tends to degrade performance.

Since there appears to be no way to circumvent the fundamental precision-performance tradeoff, we propose the following two tactics for designing synchronization algorithms for replication systems:

(a) Push the precision-performance curve as far away from the origin as possible for a given environment.

(b) Offer convenient mechanisms for controlling the point of system operation on the tradeoff curve.

Tactic (a) leads us to focus primarily on push-based approaches to replica synchronization, because they offer the opportunity for the best precision-performance curves, *i.e.*, more efficient use of communication resources, compared with pull-based approaches. (We verify this claim empirically in Chapter 2.) The reason for this advantage is that sources are better equipped than the central repository to make decisions about synchronization actions since they have access to the data itself and can obtain accurate precision measurements. Furthermore, our work uses metrics of replica precision that are based on content rather than solely on metadata. Metadata based metrics have been used to drive synchronization policies in approximate replication environments (see Section 1.5 covering related work), and are usually intended to emulate an underlying metric based on content. For example, metadata metrics based on temporal staleness or number of unreported updates reflect an attempt to capture some notion of the degree of change to the content. If an appropriate content-based metric is used instead, precision can be measured directly, potentially leading to higher quality synchronization and precision-performance curves farther from the origin, as desired.

Tactic (b) leads us to study two ways to offer users or applications control over the position of system operation on the precision-performance tradeoff curve:

1. Users specify a minimum acceptable performance level (*i.e.*, fix a y-axis position in Figure 1.2), and the replication system attempts to maximize replica precision automatically while achieving the specified level of performance.

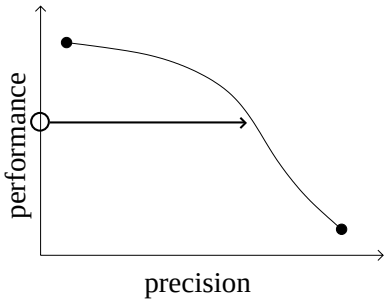


Figure 1.3: Performance Fixed/Maximize Precision.

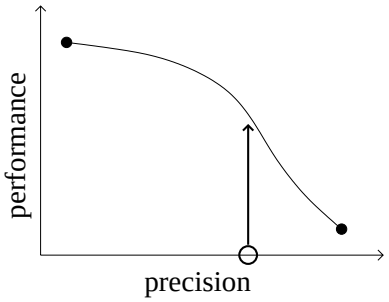


Figure 1.4: Precision Fixed/Maximize Performance.

2. Users specify a minimum allowable precision level (*i.e.*, fix an x-axis position in Figure 1.2), and the replication system attempts to maximize performance while meeting the precision requirement provided.

In each of these complementary strategies, the user (or application) fixes the position of system operation along one dimension (precision or performance), and the system is expected to maximize the position along the other dimension. These converse strategies for establishing a point of system operation on the precision-performance curve are illustrated abstractly in Figures 1.3 and 1.4, respectively.

In this dissertation we study the two strategies introduced here for controlling the operating point of a replication system in terms of precision and performance. For each strategy, we propose methods by which users or applications may constrain one dimension of the precision-performance space, and study algorithms for maximizing the position along the other dimension. Our solutions for these two complementary

strategies form the basis for offering a resource-efficient, user-controllable tradeoff between precision and performance in replication environments. We call our approach to approximate replication TRAPP (Tradeoff in Replication Precision and Performance).

1.3 Problem Statement

We now define the dual problems of maximizing precision or performance in more detail. Recall that master copies of data objects are maintained at one or more distributed data sources (Figure 1.1). Consider a data object whose master source copy O undergoes updates over time. Let $R(O)$ represent the (possibly imprecise) replica of O at the central repository. Let $V(O,t)$ represent the value of O at time t. The value of O remains constant between updates. Let $V(R(O),t)$ represent the value of $R(O)$ at time t. Object O can be *refreshed* at time t_r, in which case a message is sent to the repository, and the replicated value is set to equal the current source value: $V(R(O),t_r) = V(O,t_r)$.

For simplicity, we assume that the communication latency between sources and the central repository is small enough to be neglected. However, the techniques developed in this dissertation can tolerate nonnegligible latencies, as discussed later. We do assume there is adequate space at the central repository to store all replicas of interest. We also suppose that nodes are at all times connected to the network, and that the infrastructure is robust, *i.e.*, that node failures, network partitions, etc. are infrequent. Coping with failures or disconnections in the context of this work is outside the scope of this dissertation.

1.3.1 Maximizing Precision

For strategy (1) in Section 1.2, the goal is to maximize replica precision. Precision is quantified using a simple and general metric called *divergence*: The divergence between a source object O and its replicated copy $R(O)$ at time t is given by a numerical function $D(O,t)$. When a refresh occurs at time t_r, the divergence value

is zero: $D(O, t_r) = 0$. Between refreshes, the divergence value may become greater than zero, and the exact divergence value depends on how the master source copy relates to the replica. There are many different ways to measure divergence that are appropriate in different settings, as we discuss further in Chapter 2.

For the purpose of measuring overall divergence in the central repository, we associate with each data object O a numeric weight W_O that can be determined using a variety of criteria such as importance or frequency of access. The objective of strategy (1) is to minimize the weighted sum of the time-averaged divergence of each object, $\sum_O [W_O \cdot \int_t D(O, t)\, dt]$, under constraints on communication resources.

Communication resources may be limited at a number of points. First, the capacity of the link connecting the central data repository to the rest of the network, the *repository-side* bandwidth, may be constrained. Second, the capacity of the link connecting each source to the rest of the network, the *source-side bandwidth*, may also be constrained and may vary among sources. Moreover, all bandwidth capacities may fluctuate over time if resource limitations are related to traffic generated by other applications. We assume a standard underlying network model where any messages for which there is not enough capacity become enqueued for later transmission.

1.3.2 Maximizing Performance

The objective of strategy (2) in Section 1.2 is to maximize performance by minimizing the total *communication cost* incurred during a period of time in which replica precision is constrained. Each message sent between object O's source and the central repository (such as a refresh message) incurs a numeric cost $C_O \geq 0$, and costs are additive.

Constraints on precision arise with respect to *queries*, which are submitted by users or applications in order to access data. We consider situations in which aggregation queries over numeric data objects are submitted to the central repository. The repository is to provide answers to queries in the form of numeric intervals $[L, H]$ that are guaranteed to contain the precise answer V that could be obtained by accessing current master source copies, *i.e.*, $L \leq V \leq H$. As discussed later, guaranteed answer

intervals can be produced by establishing bounds on replica divergence. Each query submitted at the central repository specifies a *precision constraint* that specifies the maximum acceptable width ($H - L$) for the answer interval. Two modes of querying are considered in this dissertation. *One-time queries* request the answer a single time, and do not persist once the (approximate) answer has been produced. By contrast, *continuous queries* are ongoing requests for continually updated answers that meet the specified precision constraint at all times.

1.4 Organization of Dissertation

The rest of this dissertation is organized as follows:

Chapter 2: Maximizing Precision We begin by tackling the problem of maximizing replica precision when communication performance is limited due to constraints placed by users, applications, or the network infrastructure. The first step is to provide a general definition of precision, based on replica divergence, that can be specialized to a variety of data domains. We then present a replica synchronization technique whereby sources prioritize data objects that need to be synchronized based on precision considerations, and push synchronization messages to the central repository in priority order. The rate at which each source sends synchronization messages is regulated adaptively to ensure that the overall message rate conforms to the performance constraints. We evaluate our technique empirically and compare its effectiveness with that of a prior pull-based approach.

Chapter 3: Maximizing Performance Next we tackle the inverse problem: maximizing communication performance while maintaining acceptable levels of data precision as specified by users or applications at the granularity of queries over groups of objects. We focus on continuous queries, or CQ's for short, and propose a technique for performing push-based replication that meets the precision constraints of all continuous queries at all times. Without violating any query-level precision constraints, our technique continually adjusts the precision of individual replicas to maximize the overall communication performance. Results are provided from several experiments

evaluating the performance of our technique in a simulated environment. In addition, we describe a test-bed network traffic monitoring system we built to track usage patterns and flag potential security hazards using continuous queries with precision constraints. Experiments in this real-world application verify the effectiveness of our CQ-based approximate replication technique in achieving low communication cost while guaranteeing precision for a workload of multiple continuous queries.

Chapter 4: Answering Unexpected Queries Providing guaranteed precision for a workload of continuous queries does not handle an important class of queries that arises frequently in practice: one-time, unanticipated queries. Users or applications interacting with the data repository may at any time desire to obtain a one-time result of a certain query, which includes a precision constraint and may be different from the continuous queries currently being evaluated, or the same as a current CQ but with a more stringent precision constraint. Due to the ad-hoc nature of one-time queries, when one is issued the data replicas in the repository may not be of sufficient precision to meet the query's precision constraint. To obtain a query answer of adequate precision it may be necessary to access master copies of a subset of the queried data objects by contacting remote sources, incurring additional performance penalties. We study the problem of maximizing performance in the presence of unanticipated one-time queries, and devise efficient algorithms for minimizing accesses to remote master copies.

Chapter 5: Managing Precision for One-Time Queries A significant factor determining the cost to evaluate one-time queries with precision constraints at a central data repository is the precision of data replicas maintained in the repository. We study the problem of deciding what precision levels to use when replicating data objects not involved in continuous queries but subject to intermittent accesses by one-time queries. Interestingly, this problem generalizes a previously studied problem of deciding whether or not to perform exact replication of individual data objects. We propose an adaptive algorithm for setting replica precision with the goal of maximizing overall communication performance given a workload of one-time queries. In an empirical study we compare our algorithm with a prior algorithm that addresses the

less general exact replication problem, and we show that our algorithm subsumes it in performance.

Chapter 6: Visual User Interfaces In the final chapter we address some user interface issues related to approximate replication. A common method of end-user data analysis is the use of automated *data visualization* techniques: computed-generated charts and graphs that may permit various forms of user interaction. Most visualization methods assume that data is exact, while data obtained by executing continuous or one-time queries in an approximate replication system may not always be exact. In particular, using our replication techniques, queries with precision constraints over numeric data result in approximate answers in the form of numeric intervals. We propose systematic means for adapting some standard data visualization methods to handle interval approximations gracefully, with the goal of not misleading end-users regarding data precision. We also discuss the possibility that users be permitted to specify and adjust precision constraints for continuous queries over approximate replicas directly in the visualization interface. End-user control over precision constraints uncovers the important issue of motivating users to request only as much precision as they need.

1.5 Related Work

We now cover previous work that is broadly related to this dissertation. Other work not covered here is related to fairly specific aspects of this dissertation, and we cover it in the relevant chapters after describing our techniques in detail.

A vast area of research is devoted to the study of protocols to ensure *one-copy serializability* for replicated data that is updated and accessed via *transactions* [15]. A transaction is a sequence of reads and writes to data objects that must be executed to ensure certain properties. For example, each transaction is to be sheltered from the interim effects of other, concurrently executed transactions. One-copy serializability guarantees that the net effect on the replicated data of executing multiple transactions concurrently is equivalent to the effect of executing the transactions one at a time

in some order, simultaneously on all replicas. A standard example of an application domain for which transactional data access and one-copy serializability are imperative is electronic banking.

Recent work on efficient replication protocols that guarantee one-copy serializability is found in [52]. Research on transactional replication has typically assumed that each replicated data object may be read and updated from any node in a distributed environment, a more general scenario than the one covered in this dissertation. A central issue in the general setting is how to decide, for each object, which nodes should maintain replicas. The *replica placement problem* has been studied extensively; recent work in this area is found in [119].

Another critical issue in replication environments is coping with node failures and network partitions. A plethora of techniques have been proposed for ensuring one-copy serializability in the presence of failures and partitions, most based on the concept of *quorums*. Work on quorums began in the late 1970's [43, 106], and was the subject of a significant body of research throughout the 1980's and into the 1990's, *e.g.*, [4, 5, 10, 35, 51, 77, 88]. Recently, the emergence of small, wireless devices has motivated the need for replication protocols that permit mobile nodes to operate on data replicas despite being entirely disconnected from the main network. By considering transactions executed on mobile nodes during disconnected periods as only *tentative*, one-copy serializability can be achieved upon reconnection [45].

Overall, significant effort has been devoted to working around various problems while always guaranteeing one-copy serializability to meet the strict requirements of applications like electronic banking. However, there are many important applications, such as the network and sensor monitoring scenarios described above, for which rigid transactional semantics are not necessary. Furthermore, transactional semantics may hinder these applications because replication protocols for ensuring one-copy serializability tend to consume significant resources and introduce long and sometimes unacceptable processing delays [45]. Therefore, in practice many applications will enforce weaker replication semantics than one-copy serializability [92].

A number of replication strategies have been proposed based on abandoning strict

transactional replication protocols that guarantee one-copy serializability, and performing asynchronous propagation of all database updates in a nontransactional fashion [32, 97], in order to reduce response time and improve availability. These approaches alleviate many of the problems associated with transactional protocols, but they do not focus on reducing communication cost except for some batching effects since all updates are propagated eventually.

The focus of this dissertation is on conserving communication cost in nontransactional environments by performing approximate replication. The need for approximate replication is perhaps most obvious in the distributed World Wide Web environment. Partly this need arises because exact consistency is virtually impossible in the presence of the high degree of autonomy featured on the Web, but also because the volume of data is vast and aggregate data change rates are astronomical. On the Web, two forms of approximate replication are currently in heavy use: Web crawling and Web caching. Mechanisms for synchronizing document replicas in Web repositories via *incremental Web crawling*, *e.g.*, [24, 34, 117], are typically designed to work within a fixed communication budget. The order in which individual documents are refreshed is determined by the crawling scheduler with the goal of optimizing the overall precision of Web document replicas in the repository. Typically, the metric of precision used is based on some notion of *temporal staleness*: the average amount of time during which Web document replicas in the repository differ in some way from the remote master copy, *e.g.*, [25]. In Web caching environments, synchronization of a set of documents selected for caching is typically driven by constraints on the temporal staleness of cached replicas (the constraints are commonly referred to as time-to-live (TTL) restrictions), and the goal is to minimize communication, *e.g.*, [27].

Web crawling can be thought of as an instance of the *Performance Fixed/Maximize Precision* scenario, while Web caching represents an instance of the inverse *Precision Fixed/Maximize Performance* scenario. Due to the high degree of autonomy present in the Web environment, solutions to these problems almost always employ pull-oriented techniques for replica synchronization. In addition, owing to the wide variety of content found on the Web, synchronization techniques usually optimize for temporal

staleness, a simple precision metric based solely on metadata. (Other metadata-based precision metrics have also been proposed, such as the number of updates not reflected in the remote replica, *e.g.*, [54].)

For replication environments in which greater cooperation among nodes is possible and more is known about the nature of the data and needs of the users, push-oriented synchronization based on richer content-based precision metrics tends to lead to more desirable results, *i.e.*, higher quality synchronization at lower cost, as discussed in Section 1.2. The CU-SeeMe video conferencing project [33] represents an interesting instance of a push-oriented synchronization approach using direct, content-based precision metrics, which focuses on the *Performance Fixed/Maximize Precision* scenario. In CU-SeeMe, refreshes to different regions of remotely replicated images are delayed and reordered at sources based on application-specific precision metrics that take into account pixel color differences. Another domain-specific approach has been proposed for moving object tracking [120], which focuses on the *Precision Fixed/Maximize Performance* scenario, or alternatively aims at maximizing an overall "information cost" metric that combines precision and performance.

Our goal is to establish generic push-oriented approximate replication strategies that exploit and expose the fundamental precision-performance tradeoff common to all environments, in a manner suitable to a wide variety of applications that rely on replication. Some initial steps toward this goal have been made by others in previous work. To our knowledge, the first proposal on this topic was by Alonso et al. [7], and recently others have extended that work, *e.g.*, [100, 124]. All of this work falls into the *Precision Fixed/Maximize Performance* category, with precision constraints specified at the granularity of individual objects. One portion of this dissertation focuses on the inverse problem of *Performance Fixed/Maximize Precision*, which to our knowledge has not been studied in a general, application-independent setting with flexible precision metrics.

The portion of this dissertation that addresses the *Precision Fixed/Maximize Performance* problem departs significantly from previous work by considering precision constraints at the granularity of entire queries rather than at the granularity of individual replicated objects. The rationale for this choice is twofold. First, we sought to

align the granularity of precision constraint specification with the granularity of data access. (Since queries may be posed over individual data objects, our mechanism generalizes the previous approach.) Second, precision constraints at the per-query granularity leave open the possibility for optimizing performance by adjusting the allocation of precision requirements across individual objects involved in large queries. Indeed, much of our work focuses on realizing such optimizations using adaptive precision-setting techniques, which we will show to enable significant improvements in synchronization efficiency. Our work is also unique in focusing on user control over query answer precision, which may lead to unpredictable precision requirements. Specifically, to our knowledge it is the first to consider the problem of efficiently evaluating one-time queries with user-specified precision constraints that may exceed the precision of current replicas, thereby requiring access to some exact source copies.

Chapter 2

Maximizing Precision when Performance is Constrained

2.1 Introduction

In this chapter we study the problem of maximizing the precision of replicas in the presence of constraints on communication cost. Constraints may be imposed by users, applications, or the environment, which limit the availability of network or computational resources for performing replica synchronization. We focus on constraints on communication resources, but our techniques apply more generally to other types of resource limitations. In the presence of bandwidth limitations it may not be possible to maintain data object replicas at the central repository (recall Figure 1.1 in Chapter 1) always exactly synchronized with master source copies. (We assume that the communication latency between sources and the central repository is small enough to be neglected and treat replicas that would be fully synchronized if the communication latency is factored out as being exactly synchronized, for our purposes.) When exact synchronization is prevented due to bandwidth limitations, exact precision of data replicas modulo communication delays cannot be guaranteed. Instead it is desirable to maximize replica precision by minimizing the inconsistency between data in the repository and the remote source data. If users or applications access data replicas in the repository at unpredictable times, then a suitable objective is to minimize

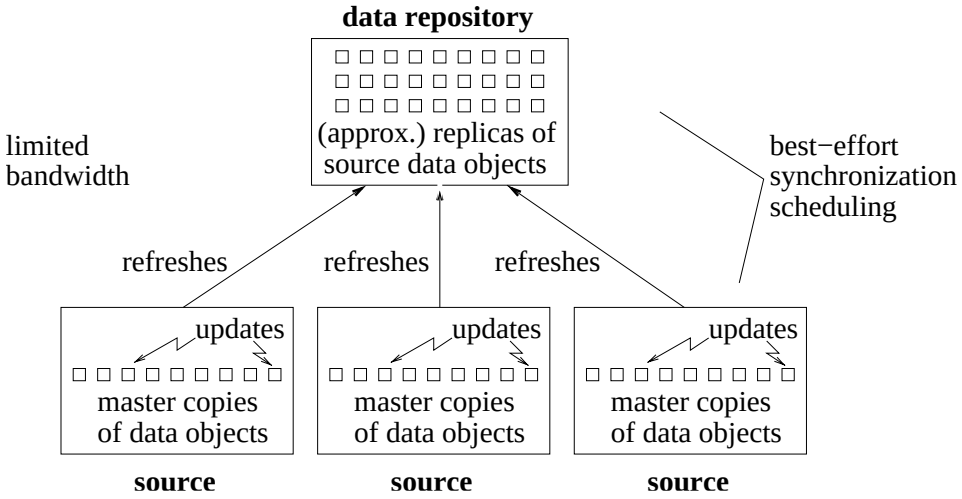


Figure 2.1: Generic framework for best-effort synchronization.

inconsistency averaged over time.

We use the term *best-effort synchronization* for the process of maintaining the repository as close as possible to exactly synchronized with the sources at all times, in the presence of limited bandwidth resources. A generic framework for best-effort synchronization is illustrated in Figure 2.1. Sources contain master copies of data objects that are updated locally. Object updates are not directly visible to the central repository, which maintains replicas that must be synchronized explicitly with source copies. The basic synchronization action is *refreshing*, in which a source sends a message to the repository containing information with which to update one or more replicas to match the current source copies. We assume the data repository contains replicas of all source objects of interest (or data derived from source objects, such as an *index*), and we deal only with the problem of keeping the values of the replicated objects as up-to-date as possible in the face of bandwidth limitations.

2.1.1 Chapter Outline

This chapter is structured as follows. The rest of Section 2.1 provides an overview of our approach to best-effort synchronization. In Sections 2.2 and 2.3 we model and analyze the problem formally to form the basis of our solution, which relies on

prioritization of synchronization actions. In Section 2.4 we present a distributed adaptive algorithm for priority-based scheduling of synchronization actions that regulates communication while attempting to maximize replica precision. We then evaluate our algorithm empirically in Section 2.5. Next, in Section 2.6 we propose ways to achieve cooperation among nodes in competitive environments. We propose techniques for reducing the processing overhead of our algorithm on sources in Section 2.7, and discuss ways to provide guarantees on divergence under certain conditions in Section 2.8. Finally, we discuss related work in Section 2.9 and summarize the chapter in Section 2.10.

2.1.2 Synchronization Scheduling and Source Cooperation

In best-effort synchronization, some policy for *synchronization scheduling* determines when replicas of data objects should be refreshed. (Remember we are assuming that due to limited resources it is not possible to refresh every object on every update.) In most refresh scheduling policies, *e.g.*, [16, 25], the data repository plays the central role: refreshes are scheduled entirely by the repository and implemented by polling the sources, without sources participating in the scheduling. These policies must try to predict which source data objects have changed, and by how much [25, 40]. If source data objects do not behave in predictable ways, the refresh schedule is likely to result in poor synchronization. Since the best synchronization policy obviously depends on how source data objects change, improved synchronization can be achieved through some level of source participation in the refresh scheduling process, as discussed in Section 1.2.

Aside from enabling better synchronization between sources and the repository, there are other, more practical, advantages of source cooperation in synchronization scheduling. First, sources can have a say in weights given to different data objects when prioritizing them for refresh. Moreover, sources can exercise control over the portion of their own bandwidth devoted to replica synchronization, *e.g.*, giving priority to servicing local user queries as they occur and participating in replica synchronization with any spare bandwidth. In contrast, synchronization policies determined

entirely by the central repository can easily underutilize available source bandwidth, leading to poor synchronization, or overutilize source bandwidth, causing a degradation of local processing. This problem is exacerbated when the resources available for synchronization fluctuate over time, *e.g.*, due to sharing network bandwidth, CPU cycles, or disk I/O's with bursty user requests.

2.1.3 Overview of Approach

We study the problem of best-effort synchronization assuming source cooperation with the goal of maximizing replica precision in the presence of constraints on bandwidth resources. We focus on approximate replication environments with a large number of sources that synchronize their data with a shared repository. (Recall that we assume the repository contains replicas or derivations of all data objects of interest, *i.e.*, we are not considering cache replacement algorithms.) The bandwidth resources for replica synchronization may be limited at a number of points. First, the capacity of the link connecting the central data repository to the rest of the network, the *repository-side* bandwidth, may be constrained. Second, the capacity of the link connecting each source to the rest of the network, the *source-side bandwidth*, may also be constrained and may vary among sources. Moreover, all bandwidth capacities may fluctuate over time if resource limitations are related to traffic generated by other applications. We assume a standard underlying network model where any messages for which there is not enough capacity become enqueued for later transmission.

While we cast our approach as coping with limited network resources (bandwidth), our ideas apply more generally when other types of resource limitations are present. For example, sources may have limited computational resources available for replica synchronization due to local processing load. Data repositories also may have limited resources for incorporating updates, especially if they perform expensive processing such as data cleaning, aggregation, or index maintenance. Accounting for the consumption of processing resources at the central repository by tasks such as these may require different cost models than the one we use here and is therefore left as a topic of future work.

Divergence and Prioritizing Refreshes

In approximate replication, the value of an object replica at the central repository may differ from the master value at the source. This difference is called *divergence*, and the goal of maximizing precision is equivalent to minimizing divergence. Divergence is a convenient metric for quantifying the degree of synchronization: Synchronization has the property that it is bounded in one extreme (exact synchronization) and unbounded in the other. Therefore, numerically a divergence metric can range over the nonnegative reals, with zero divergence representing exact synchronization and larger divergence values representing less than perfect synchronization. The precise scale depends on the divergence metric used, discussed next.

Divergence can be measured using a number of possible metrics including Boolean freshness (up-to-date or not), number of changes since last refresh, or value deviation. (We define these metrics formally in Section 2.2.1.) The best metric to use depends on the data and the replication objectives. Regardless of the divergence metric used, the goal in best-effort synchronization is to maximize overall repository precision by minimizing the (weighted) sum of the divergence values for each master source copy of a data object and its replica in the repository. Weights may be assigned to give certain objects preferential treatment based on criteria such as importance or frequency of access. Regardless of the weights used, the ideal situation of maximal overall repository precision occurs when the sum of divergence equals zero. The choice of divergence metric and weighting scheme should reflect the objectives of the replication environment in terms of repository precision, since those parameters directly affect the synchronization policy. We will revisit these issues in detail later in this chapter.

If enough resources are available it is possible to achieve near-zero overall divergence. In environments with limited resources, since not all changes can be propagated, refreshes should be prioritized based on the divergence metric and weighting scheme. Surprisingly, we will see that prioritizing refreshes based solely on the weighted divergence between master copies and replicas of data objects does not generally lead to good refresh schedules. We establish a priority policy that achieves much better synchronization. We will describe and justify our policy in Sections 2.2

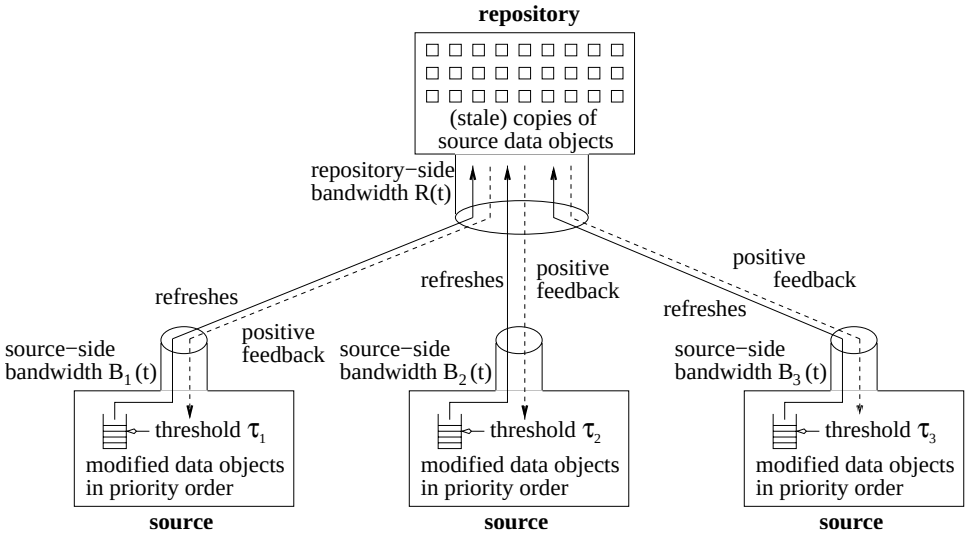


Figure 2.2: Our approach to best-effort synchronization.

and 2.3, respectively.

Coordinating Refreshes Across Multiple Sources

When multiple sources are synchronizing their objects with a shared repository, as in Figure 2.1, they must share refresh resources such as repository-side bandwidth. Hence, refreshes should be prioritized across all the sources. In the kinds of environments we are considering, sources are not typically aware of the state of the content at other sources. Furthermore, no single entity can keep track of the overall priority order across a large number of sources without incurring considerable communication overhead.

We propose a simple and effective algorithm for scheduling refreshes from a large number of sources that incurs low communication overhead while achieving synchronization that closely follows a global priority order. Our general approach is illustrated in Figure 2.2. The idea is for each source to prioritize its own modified objects locally based on the overall priority policy. Ideally, as we will see later, all modified objects having priority above a global *refresh threshold* $\mathcal{T}$ should be refreshed. However, since the best refresh threshold $\mathcal{T}$ varies over time due to fluctuating available bandwidth and divergence rates, measuring the best value for $\mathcal{T}$ and broadcasting it to all sources

is impractical, especially when the number of sources to coordinate is very large and bandwidth is limited. Consequently, each source must maintain its own independent copy of the refresh threshold, and some protocol for loosely regulating the individual thresholds needs to be in place.

One way to regulate and coordinate the source refresh thresholds without incurring too much communication overhead is to rely on occasional feedback messages from the repository requesting that sources raise or lower their thresholds. Relying on negative feedback messages from the repository to raise thresholds (in order to reduce the refresh rate) is dangerous since network resources are already overutilized, so unrecoverable network flooding situations can result. Instead, we propose an *adaptive* threshold-setting algorithm based on positive feedback. In our algorithm, sources by default gradually increase their thresholds, to conservatively reduce the refresh rate in case there is not enough bandwidth. If the repository detects a surplus of bandwidth, it sends positive feedback messages instructing sources to decrease their thresholds thereby increasing the overall refresh rate to fill the surplus.[1]

A detailed presentation and justification of our threshold-setting algorithm is given in Section 2.4. In Section 2.5, we show experimental evidence that our algorithm achieves low overall divergence without incurring excessive communication overhead, even in environments with a large number of sources and fluctuating resources and data update rates. We also demonstrate quantitatively the advantages of source cooperation in refresh scheduling over having the repository determine the synchronization schedule unilaterally as in [25].

Making Cooperation Appealing

A global priority policy, as we have been assuming, may not be realistic in environments where sources do not agree on the same policy for refresh priority. Moreover, a repository may have criteria for what to maintain up-to-date that conflicts with the objectives of some sources, *e.g.*, when the sources and repository belong to different administrative domains as is common on the Web. In Section 2.6 we describe how to

[1] We differ from the control theory use of feedback terminology, but we feel that "positive feedback" is a good term for increasing the refresh rate.

extend our synchronization techniques to reconcile the potentially different objectives among sources and between sources and the repository.

Since participating in refresh scheduling may be taxing on the computational resources of the sources, in Section 2.7 we outline lightweight mechanisms for sources to monitor the priorities of modified data objects and schedule refreshes. Techniques for incorporating changes propagated from sources into a repository without disrupting computation at the repository have already been proposed in, *e.g.*, [2, 3].

Bounding Divergence

In Section 2.8 we propose a way to provide guaranteed upper bounds on divergence in certain environments. We present a synchronization scheduling policy that minimizes the average upper bound on divergence to suit applications that require strict guarantees. By contrast, the rest of this chapter addresses the related but distinct problem of minimizing the actual divergence, whose value may be unknown to applications accessing replicated data.

2.2 Basis for Best-Effort Scheduling

In this section, we begin by formalizing our notion of divergence, then use the formal definition as a basis for a priority policy for best-effort synchronization scheduling.

2.2.1 Divergence

Consider a data object whose master source copy O undergoes updates over time. Let $R(O)$ represent the (possibly imprecise) replica of O. Let $V(O,t)$ represent the value of O at time t, and let $\nu(O,t)$ represent the *version number* of O at time t. The version number of O is incremented by one each time the value of O is updated; the value and version number of O remain constant between updates. Let $V(R(O),t)$ represent the value of $R(O)$ at time t, and similarly let $\nu(R(O),t)$ represent the version number of $R(O)$ at time t. Object O can be *refreshed* at time t_r, in which case a message is sent to the repository, and the replicated value and version number are set to equal the current

source value and version number: $V(R(O), t_r) = V(O, t_r)$; $\nu(R(O), t_r) = \nu(O, t_r)$. (Recall our assumption that the communication latency between sources and the central repository is small enough to be neglected.)

In general, let the *divergence* between a source object O and its replicated copy $R(O)$ at time t be given by a numerical function $D(O, t)$. When a refresh occurs at time t_r, the divergence value is zero: $D(O, t_r) = 0$. Between refreshes, the divergence value may become greater than zero, and the exact divergence value depends on how the master source copy relates to the replica. There are many different ways to measure divergence that are appropriate in different settings. We define three *divergence metrics* here, but the scope of our work is not limited to these specific metrics.

1. **Staleness**: $D_s(O, t) = 0$ when $\nu(R(O), t) = \nu(O, t)$; $D_s(O, t) = 1$ when $\nu(R(O), t) \neq \nu(O, t)$.[2]

2. **Lag**: $D_l(O, t) = \nu(O, t) - \nu(R(O), t)$.

3. **Value Deviation**: $D_v(O, t) = \Delta(V(O, t), V(R(O), t))$, where $\Delta(V_1, V_2)$ can be any nonnegative function quantifying the difference between two versions of an object.

Note that the first two metrics (staleness and lag) are based entirely on metadata (version numbers). The value deviation metric captures in general any metric that is based on object content rather than metadata. As discussed above, we feel that in many cases a carefully crafted divergence metric based on content can be preferable to one based only on metadata. However, in many domains a good content-based metric is not available, or its use might be expensive or impractical. For these reasons, as well as for purposes of comparison with previous work and for completeness, we use metadata-based divergence metrics in many of our examples and experiments.

When the value deviation metric is appropriate, it usually corresponds to an application-specific function that models some cost associated with the discrepancy

[2]Staleness is the reverse of Freshness (*staleness* $= 1 -$ *freshness*), which is commonly used in the literature (*e.g.*, [25, 65]) and is a direct measure of replica precision. We use staleness so that the larger value corresponds to greater divergence.

between the data value stored at the repository and the actual data value. If the data being replicated were Web documents, for example, $\Delta(V_1, V_2)$ might be based on Information Retrieval measures such as TF.IDF vector-space similarity [98]. In the CU-SeeMe video conferencing application [33] mentioned in Section 1.5, refreshes are prioritized based on the deviation between individual regions of the recorded image and their counterparts on remote viewer screens. The CU-SeeMe value deviation function $\Delta(V_1, V_2)$ is based on the sum of the absolute value of the individual pixel differences, with an additional weight for differences that occur in nearby pixels. In other applications such as stock market monitoring that have single numerical values, the simple value deviation function $\Delta(V_1, V_2) = |V_1 - V_2|$ is often suitable. Once again, note that our techniques are independent of the exact value deviation function or divergence metric used.

2.2.2 Weights

In many applications, it is desirable to bias the synchronization policy toward refreshing certain important objects more aggressively than others. *Importance* values for objects might be assigned according to various criteria, including but not limited to data quality, content provider authority (*e.g.*, PageRank [16]), and financial considerations. Our approach is independent of the exact importance criteria, but we assume a numerical importance function $\mathcal{I}(O, t) \mapsto [0, \infty)$ that may or may not change over time. In the special case where all objects have equal importance, $\mathcal{I}(O, t) = 1$ for all objects at all times.

In addition to having differing importance, objects also may differ in the frequency with which they are accessed by users or applications. The *popularity* of an object refers to some measure of the probability of access, possibly weighted by the importance of the person or application that tends to access the data. The popularity of an object O at time t is denoted $\mathcal{P}(O, t) \mapsto [0, \infty)$. In many applications it is important to account for popularity so that scarce resources are used for synchronizing data that will be accessed frequently, maximizing the likelihood of accessing closely synchronized data [65].

From importance and popularity we derive an overall *weight* $W(O,t) \mapsto [0,\infty)$ for refresh assigned to an object O at time t:

$$W(O,t) = \mathcal{I}(O,t) \cdot \mathcal{P}(O,t)$$

There could be other multiplicative factors contributing to $W(O,t)$ besides importance and popularity, based on other aspects relevant to replica synchronization. For example, one could incorporate detailed specifications of the objectives of users as in [22]. As another example, the communication cost incurred while refreshing objects may not be uniform, due to differing sizes of objects, nonuniform distances between sources and the central repository, etc. Nonuniform refresh cost can be accounted for in the weighting scheme, but other issues may arise in the presence of nonuniform cost that our techniques do not handle directly, as discussed in Chapter 7. In this chapter we assume uniform synchronization cost for all objects. Also, for now we assume that sources and the repository agree on and are aware of the weighting scheme to be used for best-effort synchronization. In Section 2.6, we address the possibility of conflicting interests among different sources and between sources and the repository.

2.2.3 Priority Scheduling

The objective of best-effort synchronization is to minimize the (weighted) sum of the time-averaged divergence of each object, under the constraint of limited resources [25]. For the staleness divergence metric, this objective is equivalent to minimizing the (possibly weighted) probability of accessing stale data [65]. We begin by studying a theoretical situation in which all sources and the repository share knowledge about each others' state without using network resources, and sources are aware of available repository-side bandwidth. By first considering this idealized situation, we establish an "ideal" scheduling policy for best-effort synchronization, on which we can base our practical techniques.

Let us assume for the moment that each source is aware of the state of objects at all other sources. We assert that if divergence patterns of objects tend to remain consistent over reasonable spans of time, objects should be prioritized globally for

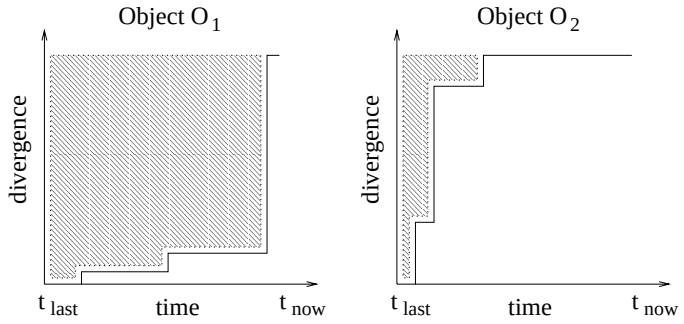


Figure 2.3: Two divergence graphs showing priority.

refreshing according to the following formula:

$$P(O_i, t_{now}) = (t_{now} - t_{last(i)}) \cdot D(O_i, t_{now}) \cdot W(O_i, t_{now}) - \int_{t_{last(i)}}^{t_{now}} D(O_i, t) \cdot W(O_i, t)\, dt$$

$P(O_i, t_{now})$ is the *refresh priority* of object O_i at time t_{now}. It is a function of the time $t_{last(i)}$ when O_i was last refreshed, the current time t_{now}, and the divergence and weight of O_i during the interval between $t_{last(i)}$ and t_{now}. The first term is the weighted product of the time interval since the last refresh and the current divergence. The subtracted term is the weighted area under the divergence curve during the interval since the last refresh. The overall priority function $P(O_i, t_{now})$ captures the area above the divergence curve between $t_{last(i)}$ and t_{now}, properly weighted.

The two graphs in Figure 2.3 depict the refresh priority for two different objects, with time on the x-axis and divergence on the y-axis. Recall that t_{last} denotes the time of last refresh. Object O_1 remained relatively unchanged until recently, then suddenly underwent a significant change. Object O_2 underwent significant changes immediately following the last refresh, but has not changed much since then. In each of the graphs, the area of the shaded region is the unweighted refresh priority for that object. Assuming the two objects are assigned same weight, O_1 will be assigned higher priority for refresh at time t_{now} than O_2.

Intuitively, higher priority is assigned when refreshing an object is likely to have more long-term benefit in terms of divergence reduction. Take object O_1 in Figure 2.3, which diverged slowly after the last refresh. Assuming it is likely to again diverge

slowly if another refresh is performed, a significant reduction in time-averaged divergence can be achieved by refreshing it immediately rather than leaving it with high divergence. On the other hand, object O_2 diverged quickly after the last refresh, so if this behavior repeats itself refreshing O_2 again is likely to have relatively little long-term benefit compared with refreshing O_1, even though they have the same current divergence. Mathematical justification and empirical validation of our refresh priority function are given in Section 2.3. In Section 7.1.1 we discuss some potential positive and negative implications of extending our priority function to take into account a longer history window.

Note that in most cases it is reasonable to assume that importance and popularity weights do not change rapidly relative to the time scale at which refreshes occur, *i.e.*, $W(O_i, t) \approx W(O_i, t_{now})$ for all $t_{last(i)} \leq t \leq t_{now}$. (In fact, in many intuitive weighting schemes, the weights are adjusted very infrequently.) Under this reasonable approximation, we can rewrite the refresh priority function as:

$$P(O_i, t_{now}) \approx \left((t_{now} - t_{last(i)}) \cdot D(O_i, t_{now}) - \int_{t_{last(i)}}^{t_{now}} D(O_i, t)\, dt \right) \cdot W(O_i, t_{now})$$

Assuming for our idealized scenario that sources know how much repository-side bandwidth is available for refreshes, the ideal synchronization schedule can be achieved as follows. Each time there is enough repository-side bandwidth to accept a refresh, the object with the highest refresh priority among all objects at all sources should be refreshed. If the source containing the highest priority object does not have enough source-side bandwidth available to perform the refresh, then the object with the second highest priority overall should be refreshed instead, and so on.

2.2.4 Special-Case Priority Functions

The refresh priority formula in Section 2.2.3 represents a general result (justified in Section 2.3), and applies to any divergence metric. We now give specialized versions of the general priority function for important special cases. The derivations of the special-case priority formulae presented here are provided in Section 2.3.2.

Consider a scenario where each object O_i is updated according to a Poisson process with parameter λ_i. In this common scenario (which has been shown to apply to Web pages [25], for example), under the staleness divergence metric specified in Section 2.2.1, the refresh priority function can be written as:

$$P_s(O_i, t_{now}) = \frac{D_s(O_i, t_{now})}{\lambda_i} \cdot W(O_i, t_{now})$$

The intuition behind this formula is quite simple. First, objects whose replicas are up-to-date have zero priority, since there is no benefit to repeatedly refreshing the same value. Among objects that are stale, it is desirable to refresh the least frequently changing ones (properly weighted), since they are the most likely to remain up-to-date the longest after being refreshed. In [25], a similar conclusion was reached for the staleness metric in high-contention scenarios. However, our result differs from the exact refresh scheduling result presented in [25], which applies to periodic pulling scenarios, because in our scenario, sources have direct knowledge of update times and decide whether to refresh immediately after each update.

Under the lag metric (recall Section 2.2.1), when updates follow a Poisson model the refresh priority function can be written as:

$$P_l(O_i, t_{now}) = \frac{D_l(O_i, t_{now}) \cdot (D_l(O_i, t_{now}) + 1)}{2\lambda_i} \cdot W(O_i, t_{now})$$

which is roughly proportional to the square of the number of updates to the source value not reflected in the remote replica. This square proportionality indicates that it is especially important to refresh objects that have undergone many changes. Moreover, the priority is inversely proportional to the average change rate λ_i. This inverse proportionality assigns higher priority to objects that are not expected to change rapidly in the future.

2.3 Justification of Refresh Priority Function

In this section we justify, both mathematically and empirically, why prioritizing objects for refreshing using the formulae proposed in Section 2.2 is appropriate for best-effort synchronization. Let us begin by assuming that bandwidth constraints restrict us to a constant B refreshes/second. (Recall that we assume all objects are of roughly equal size; in Chapter 7 we provide some discussion of nonuniform object sizes.) Say that there are a total of n objects $O_1, O_2, \cdots, O_n$ among all the data sources. Furthermore, say the divergence of each object O_i is monotonic and depends purely on the time elapsed since the last refresh: $D(O_i, t_{now}) = D^*(O_i, t_{now} - t_{last(i)})$, where $D^*(O_i, t)$ is any nonnegative function that increases monotonically with t. In this scenario, the optimal refresh schedule is one in which each object O_i is refreshed at regular intervals determined by a refresh period T_i.

To determine values for the refresh periods $T_1, T_2, \cdots, T_n$ resulting in the best refresh schedule, we must solve the following optimization problem: minimize the total time-averaged divergence $\overline{D} = \sum_{i=1}^{n}(\frac{1}{T_i} \cdot \int_0^{T_i} D^*(O_i, t)\, dt)$, subject to the bandwidth constraint $\sum_{i=1}^{n} \frac{1}{T_i} = B$. Using the method of Lagrange Multipliers [103], the optimal solution has the property that there is a single constant $\mathcal{T}$ such that for all i:

$$\Phi_i = \mathcal{T} \tag{2.1}$$

where

$$\Phi_i = T_i \cdot D^*(O_i, T_i) - \int_0^{T_i} D^*(O_i, t)\, dt$$

$\mathcal{T}$ is called the *refresh threshold*, and it controls the overall refresh rate. It corresponds to the (unweighted) priority an object must have in order to be refreshed. A small $\mathcal{T}$ value results in more refreshes, *i.e.*, a high refresh rate. A large $\mathcal{T}$ value results in a low refresh rate. The value of $\mathcal{T}$ depends on the maximum bandwidth B and how fast the objects diverge.

Interestingly, it is possible to discover the optimal refresh policy without directly solving for the refresh periods $T_1, T_2, \cdots, T_n$ if, for all $1 \leq i \leq n$, Φ_i monotonically

increases as T_i increases. Under this *monotonicity assumption*, the optimal schedule can be determined online as the current time t_{now} advances by monitoring what the value of T_i would be if object O_i were selected for refresh at the current time: $T_i = t_{now} - t_{last(i)}$. In this scheme, every object O_i would have a proposed refresh period T_i at all times. Given a proposed T_i value for object O_i, Φ_i can be computed using the relationship between t_{now}, $t_{last(i)}$, and T_i along with the relationship between $D()$ and $D^*()$. Note that we are now able to drop the assumption that objects diverge in the same manner after each refresh. We can rewrite Φ_i as the refresh priority at time t_{now}:

$$P(O_i, t_{now}) = (t_{now} - t_{last(i)}) \cdot D(O_i, t_{now}) - \int_{t_{last(i)}}^{t_{now}} D(O_i, t)\, dt \tag{2.2}$$

Thus, when an object's refresh priority reaches $\mathcal{T}$, that object should be refreshed. Under the monotonicity assumption, the refresh priority of each object monotonically increases with time, so there is exactly one point in time at which the priority equals $\mathcal{T}$, which is the optimal refresh time. By adding weights, we arrive at our original priority function in Section 2.2.3. In realistic environments, the update patterns of objects and amount of available bandwidth are likely to fluctuate over time, so the best value for the refresh threshold $\mathcal{T}$ changes as well. In Section 2.4, we give an algorithm for finding and dynamically adjusting $\mathcal{T}$ in a multiple-source environment as bandwidth and update patterns fluctuate.

2.3.1 Priority Monotonicity

We showed that if priority is expected to increase monotonically, the best time to refresh an object O_i occurs as soon as its priority reaches the refresh threshold $\mathcal{T}$. We now demonstrate that the priority of any object O_i, $P(O_i, t)$, is indeed expected to increase monotonically with time t. Taking the derivative of $P(O_i, t)$ in Equation (2.2) with respect to time, we obtain:

$$\frac{\partial}{\partial t} P(O_i, t) = (t - t_{last(i)}) \cdot \frac{\partial}{\partial t} D(O_i, t) \tag{2.3}$$

From this equation, it is easy to see that the expected value of the change in priority $\frac{\partial}{\partial t}P(O_i, t)$ is nonnegative if the expected change in divergence is nonnegative. The latter must be true over time because divergence can never become negative, therefore it must increase at least as much as it decreases. Therefore, unless some special knowledge of future update patterns indicates that an object's source value will converge back toward the replicated value, causing divergence to temporarily decrease, priority can be expected to increase monotonically over time.

2.3.2 Derivation of Special Cases

Now consider the special cases from Section 2.2.4. Recall that in those special cases each object O_i is updated according to a Poisson process with parameter λ_i. Let $u_i = \nu(O_i, t_{now}) - \nu(R(O_i), t_{now})$, meaning that there have been u_i updates to object O_i since the last refresh. The expected time elapsed since the last refresh is $t_{now} - t_{last(i)} = \frac{u_i}{\lambda_i}$.

If the lag divergence metric is used, the divergence after u_i updates without a refresh is $D_l(O_i, t_{now}) = u_i$. Immediately following the u_i-th update, the integral of divergence since the last refresh, $\int_{t_{last(i)}}^{t_{now}} D(O_i, t)dt$, is expected to equal $\frac{1}{\lambda_i} \cdot \sum_{x=0}^{u_i-1} x = \frac{u_i \cdot (u_i-1)}{2\lambda_i}$. Putting it all together, we obtain:

$$P_l(O_i, t_{now}) = \frac{u_i}{\lambda_i} \cdot D_l(O_i, t_{now}) - \frac{u_i \cdot (u_i - 1)}{2\lambda_i} = \frac{D_l(O_i, t_{now}) \cdot (D_l(O_i, t_{now}) + 1)}{2\lambda_i}$$

Using the staleness divergence metric, immediately following the u_i-th update the integral of divergence since the last refresh is expected to equal $\frac{u_i-1}{\lambda_i}$. This gives:

$$P_s(O_i, t_{now}) = \frac{u_i}{\lambda_i} \cdot D_s(O_i, t_{now}) - \frac{u_i - 1}{\lambda_i} = \frac{D_s(O_i, t_{now})}{\lambda_i}$$

2.3.3 Empirical Validation of Priority Function

As discussed in Section 2.1.3, it may appear surprising that it is not a good scheduling strategy to simply prioritize objects according to weighted divergence, *i.e.*, $P(O_i, t) = D(O_i, t) \cdot W(O_i, t)$. To validate our less intuitive priority function empirically, we

performed some simulations. We simulated a single data source containing n objects, connected to a repository with bandwidth that supports up to 10 refreshes per second. Each simulated object O_i was updated with probability λ_i each second, and upon each update, the object's value was either incremented or decremented by 1, with equal probability (following a *random walk* pattern).

In our first experiment, we set all weights to 1 and randomly assigned λ_i values to objects following a uniform distribution. We varied the number of objects from $n = 1$ to 1000 and configured the simulator to prioritize objects for refresh under each of the three divergence metrics: staleness, lag, and value deviation with $\Delta(V_1, V_2) = |V_1 - V_2|$. In all runs, the difference in overall time-averaged divergence observed between our priority function and the simpler alternative was less than 10%.

However, when we introduced some skew into the data parameters, our priority function proved to be significantly better than the simpler alternative. For example, we simulated $n = 100$ objects, a randomly-selected half of which were assigned a weight of 10 while the other half received a weight of 1. An independently- and randomly-selected half of the objects were updated with probability 0.01 while the other half were updated consistently every second. Under the staleness, lag, and deviation metrics, the simple priority function resulted in a 64%, 74%, and 84% increase in overall time-averaged divergence, respectively, compared with our priority function.

2.4 Threshold-Setting Algorithm

In Sections 2.2 and 2.3 we established our approach: prioritize objects and refresh only those whose priority is above a certain refresh threshold $\mathcal{T}$, where $\mathcal{T}$ depends on the available bandwidth and the divergence rates of the objects. Unfortunately, determining the best value for $\mathcal{T}$ would require solving a very large system of equations in most cases: one weighted instance of Equation (2.1) for each object plus an extra equation for the bandwidth constraint, assuming the bandwidth bottleneck occurs at the repository side. If repository-side bandwidth availability is not the only bottleneck, then setting $\mathcal{T}$ appropriately also requires considering restrictive source-side

bandwidth constraints, which further complicates analysis. Moreover, the available bandwidth and divergence rates may fluctuate widely over time, so most likely there is no single best threshold value that works well all the time. Even if a central site (such as the data repository) could gather all the required information and calculate $\mathcal{T}$, if $\mathcal{T}$ changes over time and communication is limited then it may be difficult or impossible to ensure that all m sources are aware of the current threshold value $\mathcal{T}$, especially if the number of sources is very large. In our approach each source S_j maintains its own local refresh threshold value $\mathcal{T}_j$. Whenever a source S_j has enough source-side bandwidth to perform a refresh,[3] it refreshes the object with the highest refresh priority if that priority is above the local refresh threshold $\mathcal{T}_j$.

As the best global threshold $\mathcal{T}$ changes over time, ideally the individual local threshold values $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m$ are maintained close to $\mathcal{T}$ to ensure the best overall synchronization schedule. We propose an adaptive algorithm in which the repository and sources work together to adjust the refresh thresholds dynamically, as was illustrated in Figure 2.2 and discussed briefly in Section 2.1.3. The desired properties of such an algorithm are threefold. First, the algorithm should cause the individual local thresholds to converge on the overall best threshold as conditions change. Second, the algorithm should incur as little communication overhead as possible so as to reserve as much bandwidth as possible for actual refreshes. Third and most importantly, the algorithm must be designed so that it is not possible for a huge excess of refresh messages to become queued in the network for a long period of time. It is crucial to avoid network flooding since refresh messages would be stalled leading to increased repository divergence.

As discussed in Section 2.1.3, the threshold-setting algorithm should avoid relying on negative feedback from the repository. Otherwise, it would be very difficult to recover from situations where the bandwidth is flooded and both refreshes and feedback messages are delayed. A more stable strategy is for the repository to send positive feedback messages when the refresh rate is too slow, asking sources to decrease their thresholds and thereby increase the overall refresh rate. In the absence of feedback,

[3] In cases where the source-side bandwidth bottleneck occurs in the communication infrastructure, surplus network bandwidth can be detected and exploited using, *e.g.*, [113]. In other cases where local processing or I/O time is the primary concern, standard scheduling techniques can be used.

sources can assume that the refresh rate is too fast and should reduce the refresh rate by increasing their thresholds.

In our algorithm, the repository continually monitors repository-side bandwidth availability.[4] If underutilized, the repository uses the excess bandwidth to send positive feedback messages to as many sources as possible (until the excess bandwidth is utilized), asking them each to decrease their thresholds by a multiplicative factor ω. If it is not possible to provide feedback to every source, the sources with the highest local thresholds are selected to receive feedback. (For the repository to track the source thresholds, each source can piggyback its current local threshold in refresh messages.) When a source S_j receives a feedback message from the repository, it decreases its local threshold $\mathcal{T}_j$ by setting $\mathcal{T}_j := \frac{\mathcal{T}_j}{\omega}$, unless it is already sending at the full capacity of the source-side bandwidth, in which case it leaves $\mathcal{T}_j$ unmodified.[5] In lieu of negative feedback, every time source S_j refreshes an object, it increases its local threshold $\mathcal{T}_j$ by a multiplicative factor $(\theta \cdot \alpha)$ by setting $\mathcal{T}_j := \mathcal{T}_j \cdot (\theta \cdot \alpha)$. Because our algorithm is adaptive, any initial values for the $\mathcal{T}_j$'s can be used and we assume a warm-up period.

The *threshold decrease parameter* ω is a constant that controls how aggressively the repository requests more refreshes. The *threshold increase parameter* θ is a constant that controls how quickly sources slow down the refresh rate in the absence of positive feedback. In Section 2.5.1 we determine good settings for these two parameters. The factor α is used to accelerate the rate of threshold increase in cases where network flooding is likely. If the elapsed time $\Delta t_{feedback}$ since the last feedback message was received at a source is less than the expected feedback period $P_{feedback}$, then $\alpha = 1$. However, whenever $\Delta t_{feedback} > P_{feedback}$, $\alpha = \frac{\Delta t_{feedback}}{P_{feedback}}$. The expected feedback period $P_{feedback}$ is estimated as the ratio of the total number of sources divided by the average

[4] Network measurement tools such as [66] can be used to monitor available network bandwidth between the repository and a gateway or backbone. Alternatively, if the repository-side bottleneck is due to local processing or I/O time as opposed to network congestion, standard scheduling techniques can be used to manage local resources.

[5] We want to avoid situations in which sources have large queues of over-threshold objects due to source-side bandwidth limitations. In such situations, if more source bandwidth suddenly becomes available, sources may flood the repository with refreshes that far exceed the repository bandwidth capacity. If, however, the repository does have plenty of bandwidth available, it will soon send positive feedback messages to the sources, triggering the right amount of additional refreshing.

repository-side bandwidth. It is not at all critical that the expected feedback period value be exact—it need only be a rough estimate.

2.5 Experimental Evaluation

We now discuss an experimental evaluation that we performed to determine good settings for the parameters ω and θ, to assess the effectiveness of our algorithm, and to compare against synchronization schedules determined by the repository alone. We constructed a discrete event simulator for an environment with one repository and m sources each containing n objects. In our simulations, the available repository-side and source-side bandwidth fluctuate over time following a sine wave pattern. The average repository-side and source-side bandwidths are controlled by simulation parameters B_C and B_S, respectively. The maximum rate of bandwidth change is controlled by simulation parameter $\Delta_m B$. When $\Delta_m B = 0$, the amount of available bandwidth remains constant. In our simulations, all messages have the same size, and each message requires 1 unit of bandwidth. For most of our experiments, we used synthetic data sets generated following a random walk as described in Section 2.3.3. Weights vary over time following sine-wave patterns with randomly-assigned amplitudes and periods. We also used one real data set, introduced in Section 2.5.2.

2.5.1 Parameter Settings

To determine the best settings for the threshold increase parameter θ and decrease parameter ω (Section 2.4), we performed a variety of simulations. We used synthetic random-walk data generated for a wide variety of configurations having up to $100,000$ objects overall, with fluctuating weights among as many as $m = 1000$ sources. We also varied the amount of repository-side and source-side bandwidth available, where both bandwidth constraints were either held constant ($\Delta_m B = 0$) or allowed to fluctuate over time at a variety of rates. We measured average divergence over a period of 5000 seconds, after an initial warm-up period.

Although our algorithm is not overly sensitive to the parameters θ and ω, it is

important to set them carefully. Setting ω too large may cause refresh messages to be sent too aggressively, thereby increasing the latency for refreshes and raising the overall divergence. However, having a small value for ω may lead to underutilization of bandwidth, which also leads to increased divergence. Setting θ too large causes sources to back off on refreshes too quickly, resulting in many positive feedback messages that reduce the bandwidth available for refreshes. On the other hand, setting θ too low sacrifices adaptiveness.

Overall, under all three divergence metrics, we found that the lowest average divergence (*i.e.*, best overall repository precision) resulted with threshold increase factor $\theta = 1.1$ and threshold decrease factor $\omega = 10$. With these settings, whenever a source refreshes an object, it increases its local threshold by 10% (or more if $\alpha > 1$ because it detects that the network seems to be flooded). Further, whenever a source receives positive feedback from the repository and it is not sending at maximum source-side capacity, it reduces its local threshold to 10% of its value. The difference in the order of magnitude between θ and ω is due to the fact that increases (due to refreshes) are much more common than decreases (due to feedback). We did not find that our algorithm was overly sensitive to the exact parameter settings (*e.g.*, $\theta = 1.2$ and $\omega = 20$ gave similar results).

2.5.2 Algorithm Effectiveness

Having determined good settings for the algorithm parameters, we ran a series of simulations comparing the divergence resulting from our algorithm with the divergence resulting from the global policy attainable only in the idealized and unrealistic scenario discussed in Section 2.2. Our comparison was performed using synthetic random-walk data where each object O_i is randomly assigned a Poisson update rate parameter λ_i. We simulated $m \in \{1, 10, 100, 1000\}$ sources, and varied the number of objects per source: $n \in \{1, 10, 100\}$, giving up to $100,000$ objects total. Objects were assigned weights randomly and weights were allowed to fluctuate over time. The average source-side bandwidth was varied between runs in $B_S \in \{10, 100\}$ and the average repository-side bandwidth was varied in $B_C \in \{10, 100, 1000, 10000, 100000\}$. Finally,

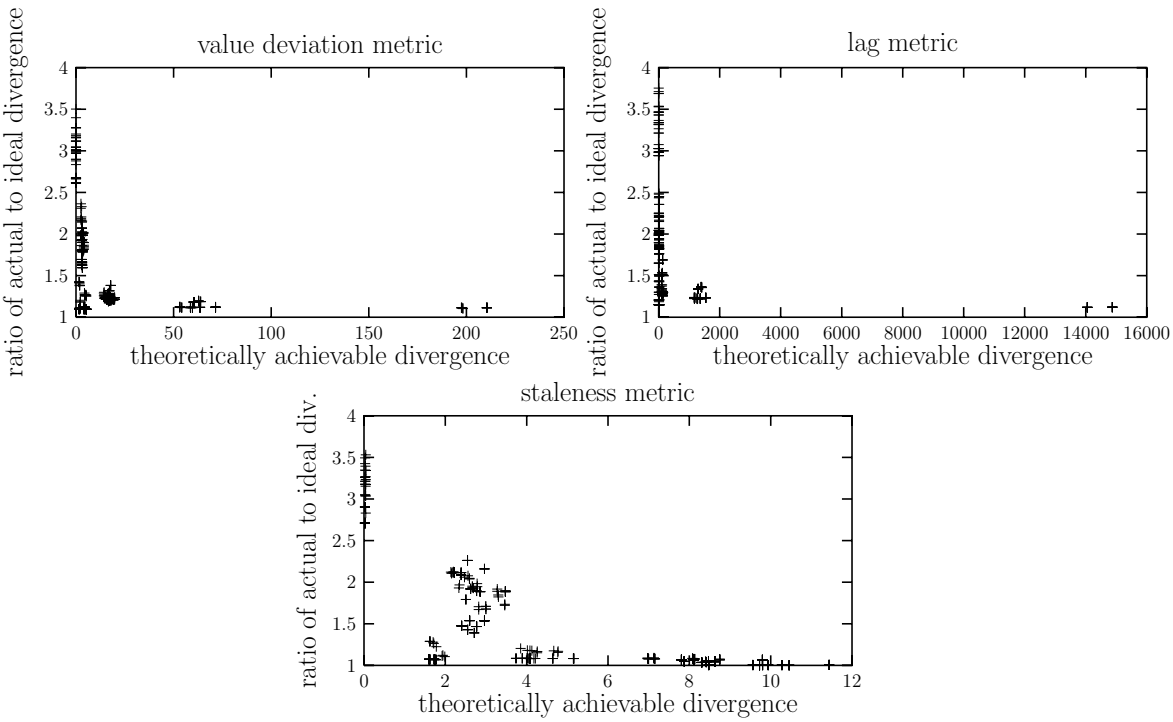


Figure 2.4: Comparison against the idealized scenario.

the bandwidth change rate was varied between runs in $\Delta_m B \in \{0, 0.005, 0.05, 0.25\}$. We measured the average divergence over a period of 5000 seconds, after an initial warm-up period.

Figure 2.4 shows the results of our experiments using the value deviation, lag, and staleness divergence metrics. One data point is plotted for every combination of the parameters described above. The y-axis shows the ratio of the average divergence resulting from our pragmatic algorithm to the average divergence theoretically attainable in the idealized scenario. Data points are arranged along the x-axis according to the theoretically attainable average divergence. The actual divergence values along the x-axis reflect the weighting scheme and vary depending on the bandwidth availability relative to the data update rates, so they are not particularly meaningful.

From Figure 2.4, we can see that as the average theoretically attainable divergence increases (due to low bandwidth and/or many rapidly diverging objects), our algorithm attains divergence nearly as good as the ideal case. On the other hand, when divergence is small, the absolute difference between the divergence achieved

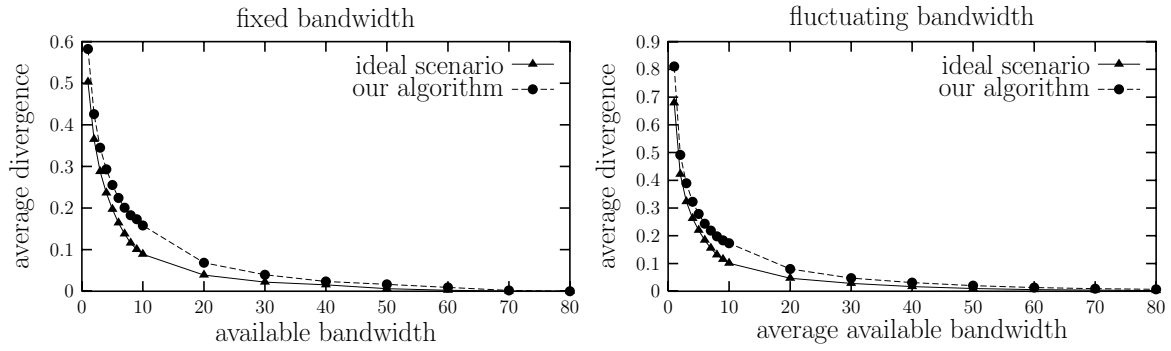


Figure 2.5: Average divergence over wind buoy data.

by our algorithm and that of the idealized case is small. Overall, our algorithm results in divergence that is close to that theoretically attainable in the idealized case. Therefore, since divergence represents the inverse of repository precision, this result suggests that our algorithm will achieve a good precision-performance curve, *i.e.*, one that is as far as possible from the origin when performance is constrained by bandwidth limitations. The remainder of Section 2.5 gives further evidence to support this claim.

Effectiveness on Real-World Data

To further verify the effectiveness of our algorithm, we performed some experiments on a real-world data set gathered from weather buoys in January 2000 by the Pacific Marine Environmental Laboratory [78]. We simulated monitoring wind vectors from $m = 40$ buoys spread out in the ocean, which perform measurements every 10 minutes. Each wind vector is made up of two numeric components, giving $n = 2$ data values per data source (buoy). All data values were equally weighted.

Using the value deviation divergence metric with $\Delta(V_1, V_2) = |V_1 - V_2|$, we simulated seven days worth of wind data, using the first day as a warm-up period. The maximum total number of messages transmitted per minute over the satellite link (repository-side bandwidth) was constrained. In the graphs in Figure 2.5, the (average) maximum bandwidth is plotted on the x-axis and the resulting average value deviation per data value is shown on the y-axis. The first graph shows the results

of experiments in which the maximum bandwidth was fixed as a constant between 1 and 80. In the second graph, available bandwidth fluctuated with time following a sine wave pattern with a peak relative change rate of $\Delta_m B = 0.25$. The wind velocity values monitored were generally in the range of 0–10, with typical values of around 5, so 0.5 on the y-axis for example indicates roughly 10% divergence. Figure 2.5 shows that the divergence achieved by our threshold-setting algorithm closely follows the divergence theoretically achievable in the idealized scenario. Figure 2.5 represents an instance of our envisioned precision-performance tradeoff curve, with both axes inverted to reflect the nature of our practical measures of repository precision and communication performance: average replica divergence and bandwidth utilization, respectively. Therefore, in this instance our algorithm successfully achieves a precision-performance curve that is at all points nearly as far from the origin as theoretically possible.

2.5.3 Comparison Against Repository-Based Scheduling

Finally, to quantify the benefits of source cooperation in synchronization scheduling, we compared our cooperative approach against a recent fully repository-driven approach by Cho and Garcia-Molina [25]. In their approach, which we will refer to as "CGM," the repository schedules all refreshes and polls sources for values. The refresh frequency for each object O_i is set independently based on an estimate of its average update rate λ_i. The goal is to minimize the staleness metric (without weights) and the overall bandwidth utilization is controlled by a numeric parameter μ, which was shown not to be solvable mathematically [25]. The CGM policy was shown to be the optimal repository-based synchronization scheduling policy, given the correct setting for μ [25]. In our experiments, we used repeated runs to experimentally determine the correct setting for their parameter μ.

Our comparison was performed over synthetic random-walk data where each object O_i is randomly assigned a Poisson update rate parameter λ_i. Since the polling model used in the CGM approach assumes no limitations on source-side bandwidth, we only placed a limitation on repository-side bandwidth, which we varied between

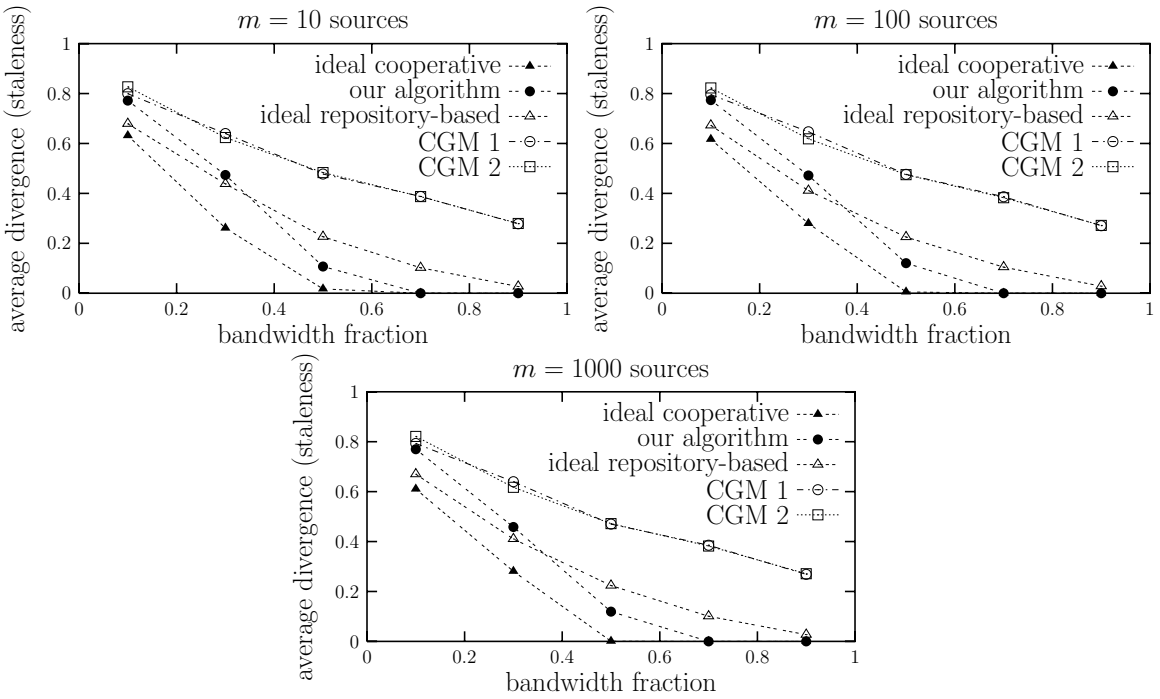


Figure 2.6: Comparison against repository-based synchronization policies.

runs. We simulated $m \in \{10, 100, 1000\}$ sources, with $n = 10$ objects per source (results for $n = 100$ objects per source were similar). We varied the bandwidth capacity between 10% and 90% of the total number of objects (*i.e.*, between $0.1 \cdot m \cdot n$ and $0.9 \cdot m \cdot n$) between runs. Since the CGM approach assumes a fixed amount of available bandwidth, this quantity was held constant during each run (*i.e.*, $\Delta_m B = 0$). We measured the average unweighted staleness over a period of 500 seconds, after an initial warm-up period. (We used a shorter measurement period in this experiment than in previous ones since the bandwidth doesn't fluctuate over time.)

Figure 2.6 shows the results of our comparison for $m = 10$, 100, and 1000 sources. In each graph, the x-axis is bandwidth capacity as a fraction of the total number of objects $m \cdot n$. The y-axis shows average divergence (staleness, in this case), and the five data lines correspond to five different theoretical or practical synchronization techniques. "Ideal cooperative" is the idealized algorithm discussed throughout this chapter, "our algorithm" is self-explanatory, and "ideal repository-based" corresponds to CGM under two theoretical assumptions: that the repository can request refreshes

without performing any communication to sources, and that the repository is aware of the exact update rates (λ values) of all of the objects. "CGM1" and "CGM2" are practical implementations of the CGM techniques. First, since refreshes require polling, each refresh incurs a round-trip message from the repository to a source. Second, the repository must estimate the object update rates (λ values) based on observations taken during prior refreshes. Two methods for estimating an object's update rate are suggested in [23]. The first method can be used if the source keeps track of the time at which the most recent update to each object occurred; this approach is CGM1. The second method for estimating update rates is used if the repository can only determine whether an object has been updated since the last refresh, but not when it was updated; this approach is CGM2.

By comparing the "ideal cooperative" and "ideal repository-based" curves in the graphs in Figure 2.6, we can see that, at least theoretically, cooperative scheduling enables much lower average divergence and therefore higher repository precision than a repository-based policy. Furthermore, by comparing the curve for our algorithm against the two pragmatic CGM curves, the attainable benefit of cooperative scheduling over repository-based techniques is demonstrated, for scenarios in which sources do cooperate.

2.6 Cooperation in Competitive Environments

So far we have assumed that there is a single priority function and refresh policy about which all participants (sources and repository) agree. However, in some environments, sources may differ in their criteria for deciding what content to keep up-to-date at a repository. Moreover, a repository's objectives of what to store and maintain up-to-date may not coincide with the goals of the sources. More concretely, the repository may request that sources implement a certain priority policy, determined by a divergence function and weights, but a given source may prefer a different priority policy derived from its own divergence function and weights. The result is that there may be two conflicting refresh priorities for each object.

As an example, consider a Web indexer, whose objective might be to focus resources on maintaining high-importance or high-popularity Web pages up-to-date in the index. Content providers' criteria for prioritizing pages for synchronization may differ from that of the indexer, and each content provider might have different criteria. For example, a retailer might wish to notify the Web indexer whenever a special offer is added to their Web site, for advertising purposes. In general, if the repository and a source disagree on the best refresh priority policy, how can a compromise be made?

Under conflicting priorities, we can partition resources among satisfying source priorities and satisfying the repository priority. Let Ψ represent the fraction of the repository-side bandwidth dedicated to satisfying source priorities, so $(1 - \Psi)$ is the fraction dedicated to repository priority. The parameter Ψ might be set by the repository administrator. In loosely coupled environments, a relatively large Ψ value can serve as an incentive for data sources to affiliate with the cooperative environment because they will be given an opportunity to keep content they value up-to-date at the repository, even if the repository prefers to focus on different content. There are at least three conceivable ways to divide up the Ψ fraction of the repository-side bandwidth dedicated to fulfilling the needs of sources:

1. All sources are given an equal share.

2. Sources are given a share proportional to the number of replicated objects from the source.

3. Sources are given a share proportional to the degree to which the source contributes to satisfying the objectives of the repository.

In options (1) and (2), all participating sources or objects are given equal treatment. In option (3), sources are allocated resources for their own purposes only if they bring significant value to the repository by offering objects that the repository wants to maintain highly synchronized. In our Web index example, in option (3) Web content providers with many documents that the index deems to be of high

value would be allocated a relatively large amount of synchronization resources to use as they see fit.

To implement options (1) or (2), the repository can monitor the total available repository bandwidth and inform sources with each feedback message how much bandwidth (in terms of number of refreshes per second) they have been allocated. Then, sources can refresh objects based on their own priority scheme at the rate specified by the repository. The remaining repository bandwidth would be dedicated to refreshes following the repository's priority, using the threshold-based algorithm proposed in Section 2.4. To implement option (3), sources would be permitted to, on average, piggyback $\frac{\Psi}{1-\Psi}$ objects of their own choosing along with every object refreshed based on the repository's priority using the threshold policy.

2.7 Priority Monitoring Techniques

In this section, we discuss some practical considerations in how sources monitor the refresh priority of their updated objects. Sources need to detect when an object's priority exceeds the refresh threshold and refresh it, assuming sufficient source-side bandwidth. If source-side bandwidth is a limiting factor, sources can maintain a priority queue so that the highest-priority updated object can be located quickly whenever spare bandwidth becomes available. We first discuss what sources need to do to compute the priority of their objects in Section 2.7.1, and then discuss when sources should measure the priority in Section 2.7.2.

2.7.1 How to Measure Priority

If the lag or staleness metrics are employed and objects are updated according to a Poisson process, then an object's priority depends uniquely on update times and not data. One simple way for the source to track priorities is to monitor when updates occur. The number of updates to an object since the last refresh determines its divergence value. The number of updates divided by the time elapsed since the last refresh gives an estimate for the Poisson parameter λ. Alternatively, the parameter

λ may be monitored over a longer period of time. From an estimate for λ and the divergence value, the refresh priority can be computed using the formulae given in Section 2.2.4. If it is impossible or too invasive to track the exact number of updates, one of the techniques proposed in [23] can be used to estimate λ. If the value deviation metric is employed, we need to compare an object's value with the older replicated value to measure its divergence, which determines the priority.

2.7.2 When to Measure Priority

Surprisingly, although the refresh priority depends on time, an object's priority can only change when an update occurs. Equation (2.3) in Section 2.3.1 shows the derivative of priority with respect to time. Note that if divergence remains constant, *i.e.*, $\frac{\partial}{\partial t}D(O_i, t) = 0$, then the priority also remains constant. Thus, an object's priority only changes when its divergence changes, which can only occur as a result of updates to the source object.

Therefore, to track the exact priority of an object, sources only need to recompute the priority when an update is made to that object. Since the priority depends on the integral of the divergence values since the last refresh, the source also needs to maintain a running total of the past divergence values weighted by the amount of time the value was active. The data necessary to compute this running total only needs to be modified each time an update occurs. Detecting updates requires the use of *triggers* [47] or a similar mechanism. If triggers are not supported or are deemed too expensive, object priority can be monitored more loosely using sampling techniques, discussed next.

Sampling for Priority

By sampling data values periodically, sources can compute divergence estimates. The current divergence of each object can be measured directly during each sample, and the sum of divergence values since the last refresh can be estimated based on past samples. Note that it is not necessary to sample at regular intervals—each sampled value can be assumed to have been active during the period beginning and ending

halfway between successive samples. Therefore, sampling can be scheduled whenever it is convenient for the source.

If the priority of an object O_i is nearing the refresh threshold, it might be appropriate to schedule the next sample of O_i based on a prediction of when the priority is expected to reach the threshold. In cases where divergence increases roughly linearly, this prediction can be made based on the rate of divergence ρ_i, which can be estimated based on previous samples.

Given an estimate for ρ_i, the projected divergence at time $t_{future} \geq t_{now}$ is $D(O_i, t_{now}) + \rho_i \cdot (t_{future} - t_{now})$. Between t_{now} and t_{future}, the integral of divergence values is projected to increase by $(t_{future} - t_{now}) \cdot (D(O_i, t_{now}) + \frac{\rho_i \cdot (t_{future} - t_{now})}{2})$. Therefore, after some algebraic simplification, the projected priority at time t_{future} is:

$$P(O_i, t_{future}) = P(O_i, t_{now}) + \frac{\rho_i}{2} \cdot (t_{future}^2 - t_{now}^2) \cdot W(O_i, t_{now})$$

By solving for t_{future}, we can determine the time at which the priority is expected to reach the refresh threshold $\mathcal{T}$:

$$t_{future} = t_{last(i)} + \sqrt{(t_{now} - t_{last(i)})^2 + \frac{2 \cdot (\mathcal{T} - P(O_i, t_{now}))}{\rho_i \cdot W(O_i, t_{now})}}$$

If a data source has extra resources available, it may make sense to schedule the next sample somewhat before that time, in case the divergence rate accelerates. The exact method used to predict the divergence rate and schedule the next sample, as well as a good choice for the regular sampling frequency, are all topics for future work.

2.8 Divergence Bounding

Some applications may require guaranteed upper bounds on the divergence of objects accessed at the repository. For example, it may be important to know with certainty that a data value is below a strict threshold or critical value. We can easily guarantee divergence bounds at the repository when the source objects have known maximum divergence rates. Let L_i be an upper bound on the total time required to refresh

object O_i.[6] Let R_i be the maximum divergence rate of object O_i. The upper bound on divergence since the last refresh at time $t_{last(i)}$ is $B(O_i, t_{now}) = R_i \cdot ((t_{now} - t_{last(i)}) + L_i)$. In applications requiring divergence bounds, it may be appropriate to perform best-effort synchronization with the goal of minimizing the upper bounds, instead of minimizing actual divergence values. Substituting $B(O_i, t_{now})$ for $D(O_i, t_{now})$ in our priority function of Section 2.2.3, we obtain the following optimal priority function for minimizing the sum of the time-averaged divergence bounds, assuming the weights do not change drastically between refreshes:

$$P(O_i, t_{now}) = \frac{R_i \cdot (t_{now} - t_{last(i)})^2}{2} \cdot W(O_i, t_{now})$$

The threshold-based algorithm from Section 2.4 for coordinating refreshes from multiple sources can be used in conjunction with this priority policy.

2.9 Related Work

A wide variety of work in the literature is related to best-effort replica synchronization to some extent. We covered some of the most relevant work in Section 1.5, and we now discuss some additional work that is specifically related to the techniques introduced in this chapter.

Theoretical algorithms for merging objects from multiple sources in priority order have been proposed in the parallel priority queue research area, *e.g.*, [17, 99]. These algorithms were designed for use in parallel computing environments with high communication throughput, and consequently require tight communication among participants. By contrast, we focus on widely distributed environments with limited communication resources. Also, network flow-control techniques such as TCP/IP have a similar flavor to our refresh coordination algorithm. However, these techniques alone are not sufficient to address our problem because they typically do not address application-level semantics such as an overall priority ranking that is independent of

[6] More generally, L_i could represent the end-to-end latency between the time a real-world event occurs, triggering a change to the source data, and the time an application reading data from the repository sees the change.

flow rates and queue sizes.

There has been a great deal of work on scheduling events in real-time systems (see [93] for a survey). Most of this work focuses on scheduling events that have strict completion deadlines, and the goal is to minimize the fraction of events that miss their deadlines. By contrast, we considered an environment in which there are no deadlines, and the goal is instead to minimize the time-average of a potentially continuous inconsistency metric.

Recent work on database-backed Web servers [65] describes strategies for ordering propagations of complex database updates to remotely cached materialized Web views in order to achieve closer synchronization. The focus of [65] is not on coping with bandwidth limitations, and it is assumed that there is adequate bandwidth to propagate all updates eventually.

2.10 Chapter Summary

In this chapter we proposed, mathematically justified, and empirically verified an algorithm for best-effort replica synchronization with source cooperation. Source cooperation in the synchronization process is advantageous for a number of reasons. First, source cooperation enables better scheduling policies than would otherwise be possible, resulting in improved synchronization (*i.e.*, better overall repository precision) compared with repository-centric approaches. Second, sources can be given a say in the relative priority of their objects for synchronization. Finally, sources can exercise fine-grained control over the source-side bandwidth used for replica synchronization so that exactly the right amount of bandwidth is available for servicing local user queries.

We began by introducing a formal objective for best-effort synchronization: minimizing the weighted sum of time-averaged divergence measures between master copies and replicas, under the assumption that data access at the central repository is entirely through local replicas. The weighting scheme provides a mechanism to tune the objective function based on potential nonuniformities in access frequencies across objects. However, it is assumed that temporal access patterns are unpredictable so,

in our objective function, divergence of each individual object is averaged uniformly over time.

After establishing our objective function, we defined and justified a priority policy for refreshing replicated objects with the goal of minimizing our objective function when bandwidth is limited due to constraints imposed by users, applications, or the environment. We then proposed an algorithm for implementing the policy, while regulating the synchronization rate to match the available bandwidth without excessive communication. Our algorithm adjusts local refresh thresholds adaptively at a large number of data sources as conditions fluctuate. We presented simulation results on both synthetic and real-world data sets to demonstrate that our techniques are effective. We also demonstrated empirically that source cooperation in synchronization scheduling leads to considerably less replica divergence and thus higher overall repository precision compared with the more conventional approach in which the repository unilaterally schedules refreshes.

Chapter 3

Maximizing Performance when Precision is Constrained

3.1 Introduction

In this and the next two chapters we focus on the scenario in which communication resources are not as heavily constrained as in Chapter 2, in which we focused on environments that have severely limited network bandwidth. As discussed in Chapter 1, even when bandwidth availability is not tightly constrained, communication may still incur a cost. The cost of communication may be manifest, for example, as a performance penalty due to increased network congestion or as monetary accountability. For applications that do not require exact precision for data objects in the repository, lower communication cost can be achieved by performing approximate replication and sacrificing some degree of replica precision.

As discussed in Section 2.8, applications can benefit from guarantees on the maximum divergence of replicas in the central repository. Divergence bounding was discussed in Section 2.8 for cases in which maximum data change rates are known. However, when communication usage is flexible, replica divergence can be bounded even without exploiting known constraints on data change rates. For an individual data object O, the divergence $D(O, t)$ can be bounded to remain below any finite, nonnegative threshold $T \geq 0$ by requiring that the source of O refresh the remote

replica $R(O)$ whenever $D(O,t) > T$. This method applies for any divergence function $D(O,t)$ and guarantees that at any time t, $D(O,t) \leq T$. (Let us assume that all messages are transmitted instantaneously and all computation is instantaneous, for now. In Section 3.5 we discuss how we handle realistic, non-negligible latencies.)

Providing guarantees of this form on the maximum divergence of individual replicated data objects is the subject of previous work, including [7, 54, 100, 124]. However, in many environments users and applications do not access data at the granularity of individual data objects. Instead, they pose *queries* over sets of replicated objects. When data access is at the granularity of queries, it is useful to obtain divergence (or equivalently precision) guarantees for query answers rather than individual objects. Furthermore, users may wish to specify a precision requirement along with each query, in terms of the maximum acceptable divergence between the answer obtained and the accurate answer that would be obtained by accessing only exact source values.

A bound on the maximum divergence of a query answer can typically be computed using a deterministic function of the divergence bounds of its inputs. For example, consider a query requesting the sum of the values of two numeric objects, O_1 and O_2, and suppose as a metric of divergence we use the numeric value deviation function $D_v(O) = |V(R(O)) - V(O)|$ proposed in Section 2.2.1. (For simplicity in this chapter we drop the time parameter t and treat it as implicitly representing the current time t_{now}.) Suppose object replicas $R(O_1)$ and $R(O_2)$ are known to diverge from their master source copies by no more than 5 and 10 units, respectively, such that $|V(R(O_1)) - V(O_1)| \leq 5$ and $|V(R(O_2)) - V(O_2)| \leq 10$. Combining these two inequalities we find that $|[V(R(O_1)) + V(R(O_2))] - [V(O_1) + V(O_2)]| \leq 15$, *i.e.*, the maximum divergence of the answer to the summation query over O_1 and O_2 is $5 + 10 = 15$, independent of the current values of O_1 and O_2.

For a given query, multiple configurations of the divergence bounds of the inputs may lead to the same overall divergence bound for the answer to the query. In the case of summation queries, for example, since maximum divergence is additive, the same divergence bound of 15 would be achieved if the divergence bounds of O_1 and O_2 were, say, 7 and 8, respectively, instead of 5 and 10. This fact holds for other types of queries as well, and leads to the following observation: Given a workload of queries

with maximum divergence requirements over replicas at the central repository, there may be multiple ways to configure the divergence bounds of individual object replicas while still adhering to the requested divergence bounds for all queries in the workload.

3.1.1 Effect of Divergence Bounds on Performance

The way individual divergence bounds are set has a significant impact on the performance of an approximate replication system in terms of overall communication cost incurred. For each replicated object, there is an inverse relationship between the allowable divergence bound and the resulting communication cost: larger divergence bounds tend to lead to lower communication cost incurred for replication, since replicas are not likely to be refreshed as frequently. The overall communication cost incurred depends both on the cost of individual refreshes and the rate at which each object must be refreshed. Thus there is an opportunity for minimizing the overall communication cost (maximizing performance) while adhering to the precision requirements of all queries by adjusting individual divergence bounds.

The optimization problem of maximizing performance subject to per-query constraints on precision is the subject of this chapter and the following two. In this chapter we study the problem of maximizing performance in the presence of a workload of continuously evaluated queries (*continuous queries*) Then in Chapters 4 and 5 we turn our attention to handling ad-hoc *one-time* queries that are evaluated only once over the current snapshot of the data rather than continuously over time. The additional challenge with one-time queries is that, when unanticipated, they may come with precision requirements that are impossible to meet using approximate replicas currently maintained at the data repository. Chapter 4 discusses how to achieve adequate precision for unanticipated one-time queries with precision constraints by accessing a minimum-cost subset of exact source copies. Chapter 5 proposes a technique for setting and adjusting divergence bounds for approximate replicas not involved in continuous queries to maximize overall performance for a workload of one-time queries.

3.1.2 Numeric Data and Aggregation Queries

Throughout Chapters 3–5 we focus on queries that perform *aggregation* over sets of numeric values. Examples of aggregation queries are the sum of the objects, the average, the minimum or maximum, etc. For both query results and individual objects we use as our divergence metric the value deviation function $D_v(O) = |V(R(O)) - V(O)|$. To be able to provide guaranteed precision for query answers, we associate with each data object O a divergence bound T_O, and require that the source of O refresh its remote replica $R(O)$ whenever it detects that $D_v(O) > T_O$. Assuming for now that message latency is negligible (we address nonnegligible message latency later), the repository can guarantee without knowing the current exact value $V(O)$ that $V(R(O)) - T_O \leq V(O) \leq V(R(O)) + T_O$.

Equivalently, the combination of $V(R(O))$ and T_O defines an interval $[V(R(O)) - T_O, V(R(O)) + T_O]$ inside which the current exact value $V(O)$ is known to lie, called the *bound* for object O. For convenience we label the interval endpoints $L_O = V(R(O)) - T_O$ and $H_O = V(R(O)) + T_O$, respectively; the bound $[L_O, H_O]$ serves as the approximate replica of object O. The source that maintains the master copy of O keeps a copy of its replicated bound $[L_O, H_O]$, and whenever the exact value $V(O)$ moves outside the bound it recenters the bound around $V(O)$ by setting $L_O := V(O) - \frac{W_O}{2}$ and $H_O := V(O) + \frac{W_O}{2}$ and sends the new bound to the data repository. The bound width $W_O = H_O - L_O$ represents twice the maximum divergence, and is inversely proportional to the precision of the approximate replica.

Approximate answers to aggregation queries can be computed over approximate replicas, *i.e.*, bounds defined in the previous paragraph. The conventional answer to an aggregation query is a single real value. We define a *bounded approximate answer* (hereafter *bounded answer*) to be a pair of real values L and H that define an interval $[L, H]$ in which the precise answer is guaranteed to lie. A *precision constraint* for a query is a user-specified constant $\delta \geq 0$ denoting a maximum acceptable interval width for the answer, *i.e.*, $0 \leq H - L \leq \delta$ at all times.

Our approximate replication architecture for achieving precision guarantees for aggregation queries over numeric data is illustrated in Figure 3.1. Users submit queries with precision constraints to the data repository, where a *query evaluator* provides

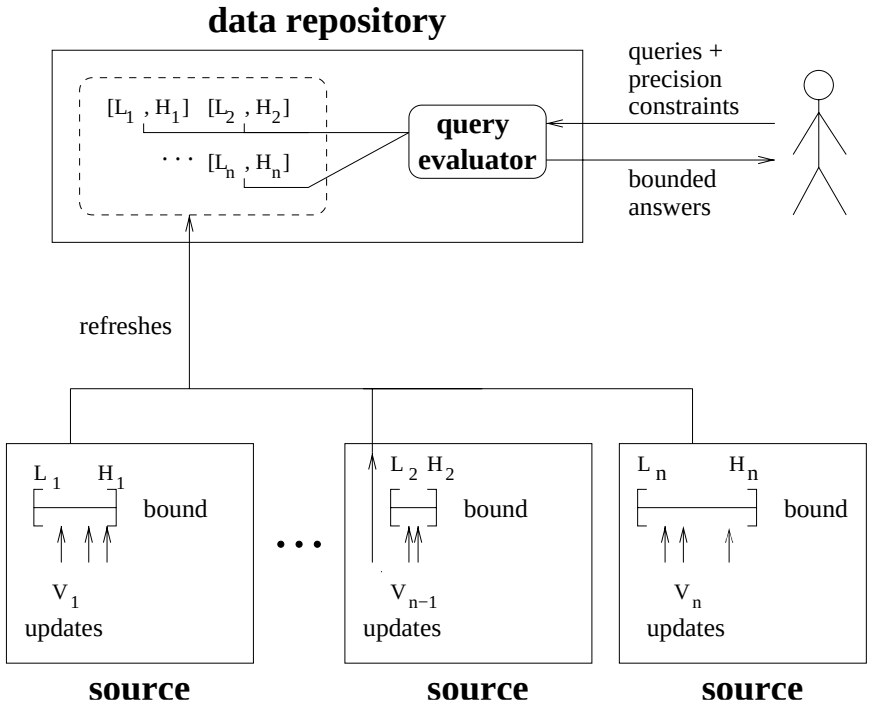


Figure 3.1: Our approximate replication architecture for achieving precision guarantees for queries over numeric data.

bounded answers computed from replicated bounds. Sources monitor updates to exact data values and refresh the replicated bounds as needed to maintain the invariant that for each object O_i, the exact value V_i is inside the bound $[L_i, H_i]$, *i.e.*, $L_i \leq V_i \leq H_i$. All insertions of new objects and deletions of existing objects are propagated immediately to the data repository, which maintains a simple catalog of all objects in the data set.

3.1.3 Example Application

One important application that relies heavily on aggregation over numeric data is *network monitoring*, which we use as a vehicle for illustrating and evaluating the techniques presented in Chapters 3 through 5. Managing complex computer networks requires tools that, among other things, report the status of network elements in real time, for applications such as traffic engineering, reliability, billing, and security, *e.g.*,

[31, 112]. Network monitoring applications do not typically require exact precision [112]. Thus, our approach can be used to reduce monitoring communication overhead between distributed network elements and a central monitoring station, while still providing quantitative precision guarantees for the approximate answers reported. Real-time network monitoring workloads often consist of a set of queries that perform numeric aggregation across distributed network elements [31, 112]. The data to be aggregated is most commonly selected or grouped by identifiers such as *source-address* and *destination-address*, or by attributes such as packet type.

3.1.4 Chapter Outline

The remainder of this chapter is structured as follows. Next, in Section 3.2 we describe some example continuous query workloads that motivate our approach to minimizing communication in the presence of continuous queries with precision constraints over approximate replicas, described in Section 3.3. Our approach relies on an adaptive algorithm for setting and adjusting the precision of individual approximate replicas, which is specified in detail in Section 3.4. Section 3.5 introduces techniques for coping with nonnegligible communication latencies. We validated our approach by building a testbed network traffic monitoring system; our implementation is described in Section 3.6, along with measurements performed to verify the effectiveness of our techniques. Related work pertaining to the *Precision Fixed/Maximize Performance* scenario studied in Chapters 3 through 5 is discussed in Section 3.7. A summary of this chapter is provided in Section 3.8.

3.2 Example Continuous Query Workloads

Familiarity with basic networking terminology [89] is assumed in this section.

Example 1: Network path latencies are of interest for infrastructure applications such as manual or automated traffic engineering, *e.g.*, [114], or quality of service (QoS) monitoring. Path latencies are computed by monitoring the queuing latency of each router along the path, and summing the current queue latencies together

with known, static transmission latencies. Since the queue latency at each router generally changes every time a packet enters or leaves the router, a naive approach could generate monitoring traffic whose volume far exceeds the volume of normal traffic, a situation that is clearly unacceptable. Fortunately, path latency applications can generally tolerate approximate answers with bounded absolute numerical error (such as latency within 5 ms of accuracy), so by using our approach obtrusive exact monitoring is avoided.

Example 2: Network traffic volumes are of interest to organizations such as internet service providers (ISP's), corporations, or universities, for a number of applications including security, billing, and infrastructure planning. Since it is often inconvenient or infeasible for individual organizations to configure routers to perform monitoring, a simple alternative is to instead monitor end hosts within the organization. We list several traffic monitoring queries that can be performed in this manner, and then motivate their usefulness. These queries form the basis of performance experiments on a real network monitoring system we have implemented; see Section 3.6.

Q_1 Monitor the volume of remote login (telnet, ssh, ftp, etc.) requests received by hosts within the organization that originate from external hosts.

Q_2 Monitor the volume of incoming traffic received by all hosts within the organization.

Q_3 Monitor the volume of incoming SYN packets received by all hosts within the organization.

Q_4 Monitor the volume of outgoing DNS lookup requests originating from within the organization.

Q_5 Monitor the volume of traffic between hosts within the organization and external hosts.

Queries Q_1 through Q_4 are motivated by security considerations. One concern is illegitimate remote login attempts, which often occur in bursts that can be detected using query Q_1. Another concern is denial-of-service (DoS) attacks. To detect the early onset of one form of incoming DoS attacks, organizations can monitor the total

volume of incoming traffic received by all hosts using query Q_2. Another form of DoS attack is characterized by a large volume of incoming "SYN" packets that can consume local resources on hosts within the organization, which can be monitored using query Q_3. Organizations also may wish to detect suspicious behavior originating from inside the organization, such as users launching DoS attacks, which may entail sending an unusually large number of DNS lookup requests detectable using query Q_4. In all of these examples, current results of the continuous query can be compared against data previously monitored at similar times of day or calendar periods that represents "typical" behavior, and the detection of atypical or unexpected behavior can be followed by more detailed and costly investigation of the data. Finally, organizations can monitor the overall traffic volume in and out of the organization using query Q_5, to help plan infrastructure upgrades or track the cost of network usage billed by a service provider.

If traffic monitoring is not performed carefully, many of these queries may be disruptive to the communication infrastructure of the organization [53]. Fortunately, these applications also do not require exact precision in query answers as long as the precision is bounded by a prespecified amount. Note that precision requirements may change over time. For example, during periods of heightened suspicion about DoS attacks, the organization may wish to obtain higher precision for queries Q_2 and Q_3 even at the cost of increased communication overhead.

3.3 Maximizing Performance for Continuous Queries with Precision Constraints

Continuous queries are registered at the central data repository, and query results are updated whenever a relevant refresh of an approximate replica is received from a remote source. Each continuous query (CQ) Q has an associated precision constraint δ_Q. We assume any number of arbitrary aggregation CQ's with arbitrary individual precision constraints. The challenge is to ensure that at all times the bounded answer to every continuous query Q is of adequate precision, *i.e.*, has width at most δ_Q, while

minimizing total communication cost. As a simple example, consider a single CQ requesting the current average of n data values at different sources, with a precision constraint δ. We can show arithmetically that the width of the answer bound is the average of the widths of the n individual bounds. Thus, one obvious way to guarantee the precision constraint is to maintain a bound of width δ on each of the n objects. Although this simple policy, which we call *uniform allocation*, is correct (the answer bound is guaranteed to satisfy the precision constraint at all times), it is not generally the best policy. To see why, it is important to understand the effects of replica bound width.

3.3.1 Effects of Bound Width

As discussed in Section 3.1.1, a bound that is narrow, *i.e.*, $H - L$ is small, enables continuous queries to maintain more precise answers due to low allowable divergence, but is likely to incur more frequent refreshes. Conversely, a bound that is wide *i.e.*, $H - L$ is large, requires fewer refreshes but causes more imprecision in query answers due to greater allowable divergence. Uniform allocation can perform poorly for the following two reasons:

1. If multiple continuous queries are issued on overlapping sets of objects, different bound widths may be assigned to the same object. While we could simply choose to use the smallest bound width, the higher refresh cost may be wasted on all but a few queries.

2. Uniform bound allocation does not account for data values that change at different rates due to different rates and magnitudes of updates. In this case, we prefer to allocate wider bounds to data values that change rapidly, and narrower bounds to the rest.

Our performance experiments (Section 3.6) that compare uniform against nonuniform bound allocation policies provide strong empirical confirmation of these observations.

Reason 2 above indicates that a good nonuniform bound width allocation policy depends heavily on the data update rates and magnitudes, which are likely to vary

over time, especially during the long lifespan of continuous queries [76]. In Example 1 from Section 3.2, a router may alternate between periods of rapidly fluctuating queue sizes (and therefore queue latencies) and steady state behavior, depending on packet arrival characteristics. Therefore, in addition to nonuniformity, we propose an *adaptive* policy, in which bound widths are adjusted continually to match current conditions.

Determining the best bound width allocation at each point in time without incurring excessive communication overhead is challenging, since it would seem to require a single site to have continual knowledge of data update rates and magnitudes across potentially hundreds of distributed sources. Moreover, the problem is complicated by Reason 1 above: we may have many continuous queries with different precision constraints involving overlapping sets of data objects. In Example 1 from Section 3.2, multiple paths whose latencies are monitored will not generally be disjoint, *i.e.*, they may share routers, and precision constraints may differ due to differences in path lengths (number of routers) as well as discrepancies in user precision requirements for different paths.

3.3.2 Adaptive Bound Width Adjustment

We have developed a low-overhead algorithm for setting bound widths adaptively to reduce communication costs while always guaranteeing to meet the precision constraints of an arbitrary set of registered aggregation CQ's. The basic idea is as follows. Each object's source shrinks the bound width periodically, at a predefined rate. Assuming the bounds begin in a state where all CQ precision constraints are satisfied (we will guarantee this to be the case), shrinking bounds only improves precision, so no precision constraint can become violated due to shrinking. The central data repository maintains a mirrored copy of the periodically shrinking bound width of each object. Each time the bound width of each object shrinks, the repository reallocates the "leftover" width to the objects at the repository it benefits the most, ensuring all precision constraints will remain satisfied.

3.3.3 Overall Approach

Our approach to adaptive precision-setting for continuous queries is illustrated in Figure 3.2, which elaborates Figure 3.1:

- *Data sources*, on the right, each store master values for one or more objects, which undergo updates over time. Sources maintain periodically-shrinking bounds for the objects. Each source forwards updates that fall outside its bound in a refresh message to the central data repository, shown on the left, and recenters that bound.
- A *precision manager* inside the data repository maintains a copy of the periodically shrinking bound width for each object. It reallocates width as described earlier and notifies the corresponding sources via *growth messages*.
- A *bound cache* inside the data repository receives all bound width changes (growth and shrinks) from the precision manager, along with all value refreshes transmitted from the data sources. The bound cache maintains a copy of the bound for each object.
- A *CQ evaluator* in the central data repository receives updates to bounds from the bound cache and provides updated continuous query answer bounds to the user.

Two aspects of our approach are key to achieving low communication cost. First, width shrinking is performed simultaneously at both the data repository and the sources without explicit coordination. Second, the precision manager uses selective growth to tune the width allocation adaptively. Informally, we minimize the overall cost to guarantee individual precision constraints over arbitrary overlapping CQ's by assigning the widest bounds to the data values that currently are updated most rapidly and are involved in the fewest queries with the largest precision constraints.

The remainder of this chapter is structured as follows:

- In Section 3.4 we specify the core of our approach: a low-overhead, adaptive algorithm for assigning bound widths to replicated objects to reduce refresh rates.

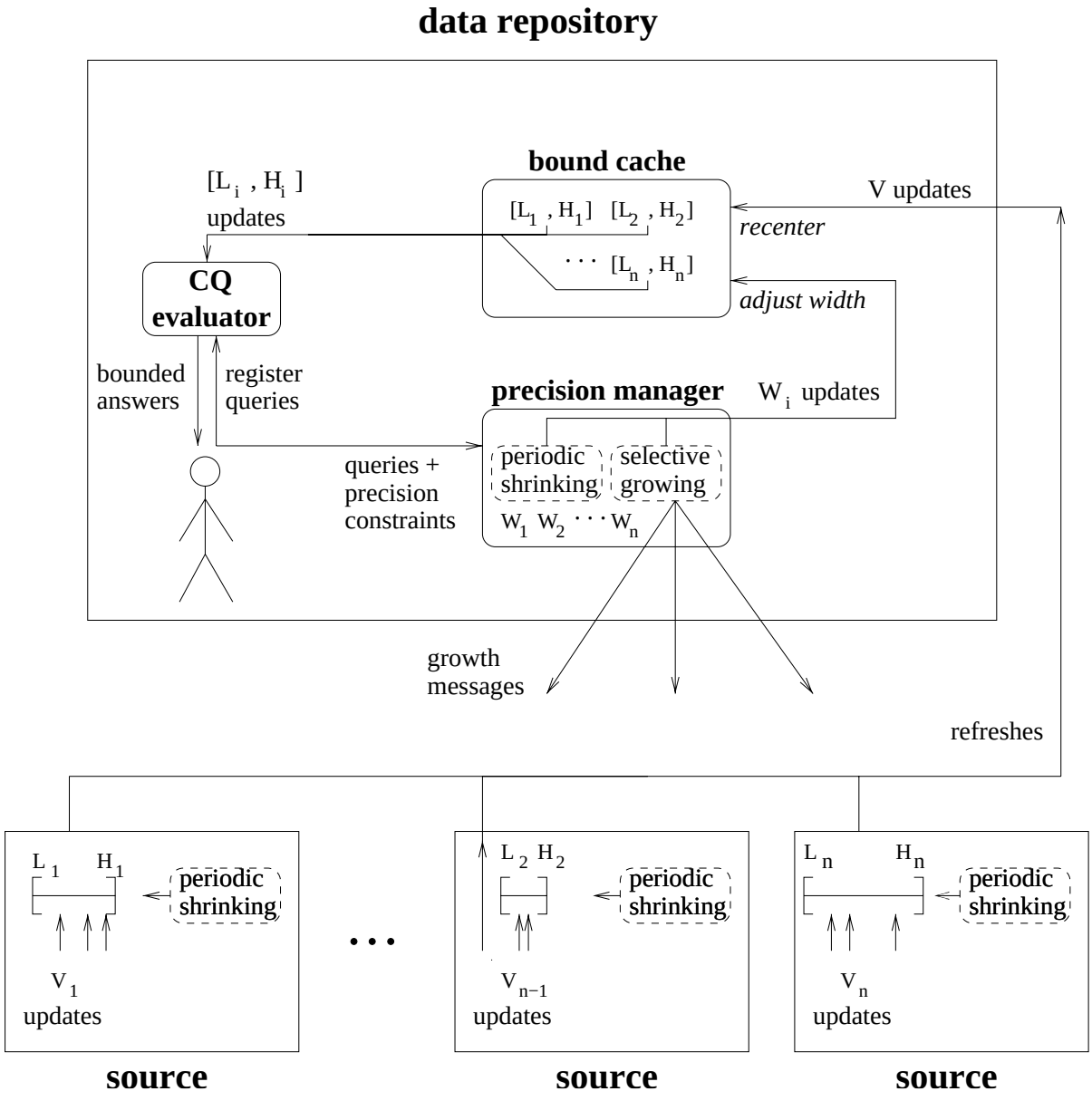


Figure 3.2: Our approach to adaptive precision-setting for continuous queries.

To guarantee adequate precision for multiple overlapping CQ's while minimizing communication, the precision manager (Figure 3.2) uses an optimization technique based on systems of linear equations.

- In Section 3.5 we describe mechanisms in the bound cache (Figure 3.2) to handle replica consistency issues that arise due to nonnegligible message latencies.
- In Section 3.6 we describe our implementation of a real network traffic monitoring system based on Example 2 in Section 3.2. We provide experimental evidence that our approach significantly reduces overall communication cost compared to a uniform allocation policy for a workload of multiple continuous queries with precision constraints.

3.4 Algorithm Description

In this section we provide more details of our approach, then we describe our algorithm for adjusting bound widths adaptively, *i.e.*, we focus on the *precision manager* in Figure 3.2 which is the core of our approach. Recall that our goal is to minimize communication cost while satisfying the precision constraints of all queries at all times. We consider continuous queries that operate over any fixed subset of the remote data values. (We do not consider selection predicates over remote values, and we assume that all insertions and deletions of new objects into the data set are propagated immediately to the central data repository.)

Queries can perform any of the five standard relational aggregation functions: COUNT, MIN, MAX, SUM, and AVG. Of these, COUNT can always be computed exactly in our setting, SUM can be computed from AVG and COUNT, and MIN and MAX are symmetric. It also turns out, as we show later in Section 3.4.4, that for the purposes of bound width setting MIN queries can be treated as a collection of AVG queries. Therefore, from this point forward we discuss primarily the AVG function. Note that queries can request the value of an individual data object by posing an AVG query over a single object.

Each registered continuous query Q_j specifies a *query set* $\mathcal{S}_j$ of objects and a *precision constraint* δ_j. (Users may later alter the precision constraint δ_j of any currently registered continuous query Q_j; see Section 3.4.6.) The query set $\mathcal{S}_j$ is a subset of a set of n data objects $O_1, O_2, \ldots, O_n$. Each data object O_i has an exact value $V(O_i)$ stored at a remote source that sends refreshes as needed to the central data repository. Say there are m registered continuous AVG queries $Q_1, Q_2, \ldots, Q_m$, with query sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m$, respectively. Then the exact answer to AVG query Q_j is $\frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \leq i \leq n, O_i \in \mathcal{S}_j} V(O_i)$. Our goal is to be able to compute an approximate answer continuously that is within Q_j's precision constraint δ_j, using cached bounds maintained by the central data repository.

Note that this goal handles *sliding window* queries [84] as well as queries over the most recent data values only. For the aggregation functions we consider, if an aggregate value is continuously computed to meet a certain precision constraint, then the result of further aggregating over time using any type of window also meets that same precision constraint. However, our algorithm does not necessarily minimize cost for sliding window queries, because sliding windows offer some leniency in the way precision bounds are set: bounds wider than δ are acceptable as long as they are compensated for by bounds narrower than δ within the same time-averaged window. Our algorithm would need to be modified to take advantage of this additional leniency in precision, which is a topic of future work.

Recall our assumption that all messages are transmitted instantaneously and all computation is instantaneous. (We discuss how to handle non-negligible latencies by modifying the bound cache in Section 3.5.) When the precision manager sends a bound growth message for object O_i to its source, or a refresh is transmitted from the source to the data repository (recall Figure 3.2), we model the cost as a known numerical constant C_i. In some scenarios the cost to refresh each object may be the same; in others it may differ due to, *e.g.*, variations in the distance of sources from the central repository. (Considering the possibility of batching refreshes from the same source is a topic of future work.) For convenience, the symbols we have introduced and others we will introduce later in this section are summarized in Table 3.1.

Before presenting our general adaptive algorithm for adjusting bound widths, we

Symbol	*Meaning*
n	number of data objects across all sources
O_i	data object ($i = 1 \ldots n$)
$V(O_i)$	exact master (source) value of object O_i
$[L_i, H_i]$	bound for object O_i
W_i	width of bound for object O_i ($W_i = H_i - L_i$)
C_i	update/growth msg. communication cost for O_i
$\mathcal{C}$	overall communication cost
λ	refresh message latency tolerance
m	number of registered continuous queries
Q_j	registered query ($j = 1 \ldots m$)
$\mathcal{S}_j$	set of objects queried by Q_j
δ_j	precision constraint of query Q_j
$\mathcal{T}$	adjustment period (algorithm parameter)
S	shrink percentage (algorithm parameter)
P_i	refresh period of O_i (last $\mathcal{T}$ time units)
B_i	burden score of O_i (computed every $\mathcal{T}$ time units)
T_j	burden target of Q_j (computed every $\mathcal{T}$ time units)
D_i	deviation of O_i (determines growth priority)

Table 3.1: CQ precision-setting model and algorithm symbols.

describe two simple cases in which the bound width of certain objects should remain fixed. First, consider an object O that is involved only in queries that request a bound on the value of O alone (AVG queries over one value). Then it suffices to fix the bound width of O to be the smallest of the precision constraints: $W_O = \min(\delta_j)$ for queries Q_j with $\mathcal{S}_j = \{O\}$. Second, for objects that are not included in any currently registered query, the bound width should be fixed at ∞ so that no refreshes for those objects are transmitted to the central data repository. The remainder of the objects, namely those that are involved in at least one query over multiple objects, pose our real challenge.

To guarantee that all precision constraints are met, the following constraint must hold for each query Q_j:

$$\sum_{1 \le i \le n, O_i \in \mathcal{S}_j} W_i \quad \le \quad \delta_j \cdot |\mathcal{S}_j|$$

In other words, the sum of bound widths for each query must not exceed the product of the precision constraint and the number of objects queried. Initially, the bounds can be set in any way that meets the precision constraint of every query, *e.g.*, by performing uniform allocation for each query, and for objects assigned multiple bound widths, taking the minimum. Then, as discussed in Section 3.3.3, our general strategy is to reallocate bound width adaptively among the objects participating in each query. Reallocation is accomplished with low communication overhead by having bounds shrink periodically over time and having the data repository's precision manager periodically select one or more bounds to grow based on current conditions. In Section 3.4.1 we describe the exact way in which bounds are shrunk in our algorithm, and then in Section 3.4.2 we describe when and how bounds are grown. In Section 3.4.5 we provide empirical validation that our algorithm converges on good bounds.

3.4.1 Bound Shrinking

Every object O_i has a corresponding bound width W_i that is maintained simultaneously at both the central data repository and at the source. Periodically, every $\mathcal{T}$ time units (seconds, for example), O_i's bound width is decreased symmetrically at both the source and the data repository by setting $W_i := W_i \cdot (1 - S)$. The constant $\mathcal{T}$ is a global parameter called the *adjustment period*, and S is a global parameter called the *shrink percentage*. The effect is to decrease the bound width by the fraction S every time unit, causing refreshes to become more frequent over time (*i.e.*, smaller refresh period). All adjustments to the bound width—decreases as well as increases—occur at intervals of $\mathcal{T}$ time units. Note that refreshes may be sent to the central data repository at any time but they simply reposition bounds without altering the width. We will discuss good settings for algorithm parameters $\mathcal{T}$ and S in Section 3.6.

3.4.2 Bound Growing

Every $\mathcal{T}$ time units, when all the bound widths shrink automatically as described in the previous section, the precision manager selects certain bound widths to grow instead, causing refreshes to become less frequent over time (*i.e.*, larger refresh period). Selecting bounds to increase (and how much) is one of the most intricate parts of our approach.

The first step is to assign a numerical *burden score* B_i to each queried object O_i. Conceptually, the burden score embodies the degree to which an object is contributing to the overall communication cost due to refreshes. The burden score is computed as $B_i = \frac{C_i}{P_i \cdot W_i}$ where recall that C_i is the cost to send a refresh of object O_i, and W_i is the current bound width. P_i is O_i's estimated *refresh period* since the previous width adjustment action, computed as $P_i = \frac{\mathcal{T}}{N_i}$ where N_i is the number of updates of O_i received by the data repository in the last $\mathcal{T}$ time units. (If $N_i = 0$ then $P_i = \infty$ so $B_i = 0$.) The burden formula is fairly intuitive since, *e.g.*, a wide bound or long refresh period reduces B_i. The exact mathematical derivation is given below in Section 3.4.3.

Once each object's refresh period and burden score have been computed, the second step is to assign a value T_j, called the *burden target*, to each AVG query Q_j. Conceptually, the burden target of a query represents the lowest overall burden required of the objects in the query in order to meet the precision constraint at all times. Since understanding the way we compute burden targets is rather involved, we present our method in full detail below, and summarize the process here. For queries over objects involved in no other queries, the burden target is set equal to the average of the burden scores of objects participating in that query. For queries that overlap it turns out that assigning burden targets requires solving a system of m equations with $T_1, T_2, \ldots, T_m$ as m unknown quantities. Because solving this system of equations exactly at run-time is likely to be expensive, we find an approximate solution by running an iterative linear equation solver until it converges within a small error ϵ. (Performance is evaluated below.)

Once a burden target has been assigned to each query, the third step is to compute for each object O_i its *deviation* D_i:

$$D_i = \max\left\{ B_i - \sum_{1 \le j \le m, O_i \in \mathcal{S}_j} T_j \ , \quad 0 \right\}$$

Deviation indicates the degree to which an object is "overburdened" with respect to the burden targets of the queries that access it. To achieve low overall refresh rates, it is desirable to equally distribute the burden across all objects involved in a given query. We justify this claim mathematically in Section 3.4.3, and we verify it empirically in Sections 3.4.5 and 3.6.

To see how we can even out burden, recall that the burden score of object O_i is $B_i = \frac{C_i}{P_i \cdot W_i}$, so if the bound $[L_i, H_i]$ were to increase in size, B_i would decrease.[1] Therefore, the burden score of an overburdened object can be reduced by growing its bound. Growth is allocated to bounds using the following greedy strategy. Queried objects are considered in decreasing order of deviation, so that the most overburdened objects are considered first. (It is important that ties be resolved randomly to prevent objects having the same deviation—most notably 0—from repeatedly being considered in the same order.) When object O_i is considered, the maximum possible amount by which the bound can be grown without violating the precision constraint of any query is computed as:

$$\Delta W_i = \min_{1 \le j \le m, O_i \in \mathcal{S}_j} \left(\delta_j \cdot |\mathcal{S}_j| - \sum_{1 \le k \le n, O_k \in \mathcal{S}_j} W_k \right)$$

If $\Delta W_i = 0$, then no action is taken. For each nonzero growth value, the precision manager increases the width of the bound for O_i symmetrically by setting $L_i := L_i - \frac{\Delta W_i}{2}$ and $H_i := H_i + \frac{\Delta W_i}{2}$. After all growth has been allocated the precision manager sends a message to each source having objects whose bound width was selected for growth.

In summary, the procedure for determining bound width growth is as follows:

1. Each object is assigned a *burden score* based on its refresh cost, estimated refresh

[1] This reasoning relies on P_i not decreasing when W_i increases, a fact that holds intuitively and is discussed further in Section 3.4.3.

period, and current bound width.

2. Each query is assigned a *burden target* by either averaging burden scores or invoking an iterative linear solver (described).

3. Each object is assigned a *deviation* value based on the difference between its burden score and the burden targets of the queries that access it.

4. The objects are considered in order of decreasing deviation, and each object O_i is assigned the maximum possible bound growth ΔW_i when it is considered.

Complexity and scalability of this approach are discussed shortly.

Burden Target Computation

We now describe how to compute the burden target T_j for each query Q_j, given the burden score B_i of each object O_i (Step 2 above). Recall that conceptually the burden target for a query represents the lowest overall burden required of the objects in the query in order to meet the precision constraint at all times. For motivation consider first the special case involving a single AVG query Q_k over every object $O_1, \ldots, O_n$. In this scenario, the goal for adjusting the burden scores simplifies to that of equalizing them (as shown mathematically in Section 3.4.3) so that $B_1 = B_2 = \cdots = B_n = T_k$. Therefore, given a set of burden scores that may not be equal, a simple way to guess at an appropriate burden target T_k is to take the average of the current burden scores, *i.e.*, $T_k = \frac{1}{|\mathcal{S}_k|} \cdot \sum_{1 \leq i \leq n, O_i \in \mathcal{S}_k} B_i$. In this way, objects having higher than average burden scores will be given high priority for growth to lower their burden scores, and those having lower than average burden scores will shrink by default, thereby raising their burden scores. On subsequent iterations, the burden target T_k will be adjusted to be the new average burden score. This overall process results in convergence of the burden scores.

We now generalize to the case of multiple queries over different sets of objects. It is useful to think of the burden score of each object involved in multiple queries as divided into components corresponding to each query over the object. Let $\theta_{i,j}$ represent the portion of object O_i's burden score corresponding to query Q_j so

that $\sum_{1 \leq j \leq m, O_i \in \mathcal{S}_j} \theta_{i,j} = B_i$. The goal for adjusting burden scores in the presence of overlapping queries is to have the burden score B_i of each object O_i equal the sum of the burden targets of the queries over O_i (as shown in Section 3.4.3). This goal is achieved if for each query Q_j over O_i, $\theta_{i,j} = T_j$. Therefore, our overall goal can be restated in terms of θ values as requiring that for every query Q_j, $\theta_{1,j} = \theta_{2,j} = \cdots = \theta_{n,j} = T_j$ (the $\theta_{i,j}$ values for objects $O_i \notin \mathcal{S}_j$ are irrelevant). Therefore, given a set of θ values, a simple way to guess at an appropriate burden target T_j for each query Q_j is by taking the average of the θ values of objects involved in Q_j, *i.e.*, $T_j = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \leq i \leq n, O_i \in \mathcal{S}_j} \theta_{i,j}$. For each object/query pair O_i/Q_j, we can express $\theta_{i,j}$ in terms of B_i, which is known, and the θ values for the other queries over O_i, which are unknown: $\theta_{i,j} = B_i - \sum_{1 \leq k \leq m, k \neq j, O_i \in \mathcal{S}_k} \theta_{i,k}$. If we replace each occurrence of $\theta_{i,k}$ by T_k for all $k \neq j$ (because we want each $\theta_{*,k}$ to converge to T_k), we have $\theta_{i,j} = B_i - \sum_{1 \leq k \leq m, k \neq j, O_i \in \mathcal{S}_k} T_k$. Substituting this expression in our formula for guessing at burden targets based on θ values, we arrive at the following expression:

$$T_j = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \leq i \leq n, O_i \in \mathcal{S}_j} \left(B_i - \sum_{1 \leq k \leq m, k \neq j, O_i \in \mathcal{S}_k} T_k \right)$$

This result is a system of m equations with $T_1, T_2, \ldots, T_m$ as m unknown quantities, which can be solved using a linear solver package.

Algorithm Complexity and Scalability

Let us consider the complexity of our overall bound growth algorithm, which is executed once every $\mathcal{T}$ time units. Most of the steps involve a simple computation per object, and the objects must be sorted once. In the last step, to compute ΔW_i efficiently, the precision manager can continually track the difference ("leftover width") between each query's precision constraint and the current answer's bound width. Then for each object we use the precomputed leftover width value for each query over that object. When queries are over overlapping sets of objects, an iterative linear solver is required to compute the burden targets, which we expect to dominate the

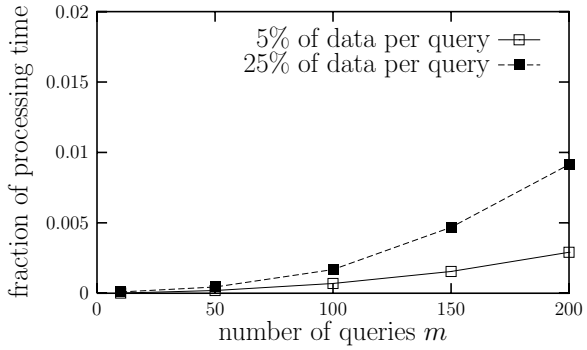


Figure 3.3: Scalability of linear system solver.

computation. The solver represents the system of m equations having $T_1, T_2, \ldots, T_m$ as m unknown quantities as an m by $(m+1)$ matrix, where entries correspond to pairs of queries. Fortunately, the matrix tends to be quite sparse: whenever the query sets $\mathcal{S}_x$ and $\mathcal{S}_y$ of two queries Q_x and Q_y are disjoint, the corresponding matrix entry is 0. For this reason, along with the fact that we can tune the number of iterations, burden target computation using an iterative linear system solver should scale well. We use a publicly-available iterative solver package called *LASPack* [101], although many alternatives exist. Convergence was generally achieved in very few iterations, and the average running time on a modest workstation was only 2.73 milliseconds in our traffic monitoring implementation using multiple overlapping queries (Section 3.6).

To test the scalability of our algorithm to a larger number of queries and data objects than we used in our implementation, we generated two sets of synthetic workloads consisting of AVG queries over a real-world 200-host network traffic data set (details on this data set are provided in Section 3.6). We treated each host as a simulated data source with one traffic level object. In one set of workloads, each query is over a randomly-selected 5% (10) of the data sources. In the second set of workloads, each query is over 25% (50) of the data sources, resulting in a much higher degree of overlap among queries. (The degree of overlap determines the density of the linear equation matrix, which is a major factor in the solver running time.) Varying the number of queries m, we measured the average running time on a Linux workstation with a 933 MHz Pentium III processor. We set the error tolerance for the LASPack

iterative solver small enough that no change in the effectiveness of our overall algorithm could be detected. Figure 3.3 shows the fraction of available processing time used by the linear solver when it is invoked once every 10 seconds (when time units are in seconds and $\mathcal{T} = 10$, which turns out to be a good setting as we explain later in Section 3.6). Allocating bound growth to handle 200 queries over 25% of 200 data sources requires only around 1% of the CPU time at the data repository.

3.4.3 Mathematical Justification for Bound Growth Strategy

Here we give a mathematical model for the behavior of numerical data objects that are replicated approximately using bounds, and we use it to justify the bound growth allocation strategy presented in Section 3.4.2. We model the behavior of approximately replicated objects as follows. For an object O_i whose exact value $V(O_i)$ varies with time, we assume that the refresh period P_i is a function of the bound width $W_i = H_i - L_i$, and signify this relationship by writing $P_i(W_i)$ instead of P_i. Intuitively, when a bound is narrow, the actual value is likely to exceed it more often and therefore the refresh period will be short. Conversely, when a bound is wide we expect the refresh period to be longer. The precise relationship between W_i and P_i depends on the behavior of $V(O_i)$.

Since each refresh of object O_i incurs a cost C_i, we can express the communication cost of the entire system as:

$$\mathcal{C} = \sum_{1 \le i \le n} \frac{C_i}{P_i(W_i)}$$

If no continuous queries are registered, then zero cost can be achieved by setting all bounds to $[-\infty, \infty]$. However, each query Q_j with precision constraint δ_j imposes the following constraint on the bound widths:

$$\sum_{1 \le i \le n, O_i \in \mathcal{S}_j} W_i \le \delta_j \cdot |\mathcal{S}_j|$$

(Recall that all basic aggregation queries can be treated as AVG queries.) We are now

faced with the optimization problem of minimizing the overall cost $\mathcal{C}$ while satisfying the above constraint for each of m queries $Q_1, Q_2, \ldots, Q_m$.

Unless the function $P_i(W_i)$ is inversely proportional to W_i, which is unlikely as discussed above, we are faced with a nonlinear optimization problem with inequality constraints. Since such problems are very difficult to solve, we decided to try treating the inequality constraints as equality constraints to get an idea of the form of the solution. (We later verified the success of this approach by comparing results obtained using the algorithm which we derive from it with results obtained by executing a nonlinear optimization problem solver that operates over synthetic data; see Section 3.4.5.) We can apply the method of Lagrange Multipliers [103] to minimize $\mathcal{C}$ under a set of m equality constraints of the form:

$$\sum_{1 \leq i \leq n, O_i \in \mathcal{S}_j} W_i = \delta_j \cdot |\mathcal{S}_j|$$

The solution [103] has the property that there are a set of m constants $\lambda_1, \lambda_2, \ldots, \lambda_m$ such that for all i:

$$C_i \cdot \frac{\partial}{\partial W_i} \left(\frac{1}{P_i(W_i)} \right) = \sum_{1 \leq j \leq m, O_i \in \mathcal{S}_j} \lambda_j$$

To evaluate the derivative we make the assumption that the function $P_i(W_i)$ has roughly the form $P_i(W_i) = Z_i \cdot (W_i)^p$, where each Z_i is an arbitrary constant and p can be any positive real number. For example, this model with $p = 2$ applies to data that follows a random walk pattern, as we now derive.

In the random walk model, after t steps of size s_i, the probability distribution of the value is a binomial distribution with variance $\sigma_i^2 = (s_i)^2 \cdot t$ [44]. Chebyshev's Inequality [44] gives an upper bound on the probability $\mathcal{P}$ that the value is beyond any distance k from the starting point: $\mathcal{P} \leq \frac{\sigma_i^2}{k^2}$. If we let $k = \frac{W_i}{2}$, treat the upper bound as a rough approximation, and solve for t when $\mathcal{P} = 1$, we obtain $t \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$, which is roughly the expected refresh period, so $P_i(W_i) \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$. In relation to our general $P_i(W_i)$ expression, for random walks $Z_i = \frac{1}{(2 \cdot s_i)^2}$ and $p = 2$.

In general, assuming $P_i(W_i)$ roughly follows the form $P_i(W_i) = Z_i \cdot (W_i)^p$ for any

$p > 0$, we can evaluate the partial derivative to obtain the following expression:

$$\frac{C_i}{P_i \cdot W_i} = M \cdot \sum_{1 \leq j \leq m, O_i \in \mathcal{S}_j} \lambda_j$$

where M is a constant. Finally, let the burden target $T_j = M \cdot \lambda_j$ and recall that the burden score $B_i = \frac{C_i}{P_i \cdot W_i}$, giving:

$$B_i = \frac{C_i}{P_i \cdot W_i} = \sum_{1 \leq j \leq m, O_i \in \mathcal{S}_j} T_j$$

According to this formula, we want P_i and W_i to be set such that the burden score B_i of each object O_i roughly equals the sum of the burden targets of all queries over O_i. Our algorithm described in Section 3.4 converges to this state by monitoring the burden scores and increasing W_i, and consequently P_i as well, to decrease B_i when it becomes significantly higher than the sum of estimated targets.

3.4.4 MIN Queries

We now consider MIN queries, and show that for the purposes of bound width setting they can be treated as a collection of AVG queries. Consider a MIN query Q_j over query set $\mathcal{S}_j$ with precision constraint δ_j. First, we show that if the bound for each object $O_i \in \mathcal{S}_j$ has width at most δ_j, the precision constraint is always met. To see this fact, observe that the answer $[L, H] = [\min(L_i), \min(H_i)]$, and it has width $H - L = \min(H_i) - \min(L_i) \leq H_k - \min(L_i)$, for the upper bound H_k of any object $O_k \in \mathcal{S}_j$. If we choose O_k to be the object with the lowest lower bound, *i.e.*, $L_k = \min(L_i)$, we obtain $H - L \leq H_k - L_k$. Thus, if all bounds have width at most δ_j, then the answer bound has width $H - L \leq \delta_j$.

We now show the converse: if the bound for some $O_i \in \mathcal{S}_j$ has width greater than δ_j, then the precision constraint cannot be guaranteed. To see this fact, consider an object $O_i \in \mathcal{S}_j$ whose value is far greater than the minimum value of the queried objects. It may seem safe to assign a bound $[L_i, H_i]$ of width exceeding δ_j to O_i, as long as L_i is greater than the lowest lower bound, since $[L_i, H_i]$ will not contribute to

the answer bound. However, if the data repository receives an update of one or more objects, causing L_i to suddenly become the lowest lower bound and H_i the lowest upper bound, then the new answer bound has width greater than δ_j. Although this situation could be remedied by requesting a tighter bound for O_i from its source, this procedure would incur a delay during which Q_j's answer bound violates its precision constraint, breaking our requirement of continuous precision for continuous queries. Therefore, for a MIN query Q_j, the bound for each queried object must have width at most δ_j at all times, and those widths are guaranteed to uphold the precision constraint. (In circumstances where occasional precision constraint violations for short periods of time are tolerable, the technique of [9], which avoids communication altogether for objects far from the current minimum, can be used instead.)

Based on the above observations, for the purposes of bound width setting, a MIN query Q_j with precision constraint δ_j over a set of objects $\mathcal{S}_j$ is equivalent to a set of single-object queries over each $O_i \in \mathcal{S}_j$ with precision constraint δ_j for each. Since single-object queries are AVG queries over one object, our bound width adjustment techniques described above can be applied to MIN queries without modification.

Overall, since SUM can be computed from AVG, and MAX is symmetric to MIN, the Section 3.4 techniques can be used for any workload consisting of a combination of SUM, AVG, MIN, and MAX queries.

3.4.5 Validation Against Optimized Strategy

We performed an initial validation of our bound width allocation strategy based on periodic shrinking and selective growing using a discrete event simulator with synthetic data. The goal of our simulation experiments is to show that our adaptive algorithm converges on the best possible bound widths, given a steady-state data set. For this purpose, we generated data for one object per simulated source following a random walk pattern, each with a randomly-assigned step size, and compared two unrealistic algorithms. In the "idealized" version of our algorithm, messages sent by the data repository to sources instructing them to grow their bounds incur no communication cost. Instead, only refresh costs were measured, to focus on the bound

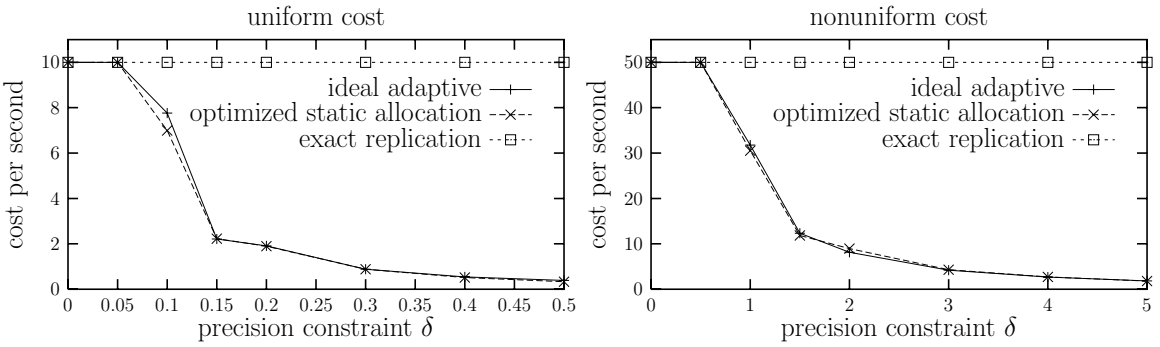


Figure 3.4: Ideal adaptive algorithm vs. optimized static allocation, random walk data.

width choices only. We compared the overall refresh cost against the refresh cost when bound widths are set statically using an optimization problem solver, described next.

The nature of random walk data makes it possible to simplify the problem of setting bound widths statically to a nonlinear optimization problem. Recall from Section 3.4.3 that the overall cost $\mathcal{C} = \sum_{1 \leq i \leq n} \frac{C_i}{P_i(W_i)}$, where $P_i(W_i)$ is the refresh period as a function of the bound width of O_i. In Section 3.4.3 we derived an approximate formula for this function in the random walk case: $P_i(W_i) \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$, which depends on the step size s_i. If the step sizes of all the objects are known, then a good static bound width allocation can be found by solving the following nonlinear optimization problem: minimize $\sum_{1 \leq i \leq n} \frac{C_i \cdot (2 \cdot s_i)^2}{(W_i)^2}$ in the presence of m constraints of the form $\sum_{1 \leq i \leq n, O_i \in \mathcal{S}_j} W_i \leq \delta_j \cdot |\mathcal{S}_j|$.

While nonlinear optimization problems with inequality constraints are difficult to solve exactly, an approximate solution can be obtained with methods that use iterative refinement. We used a package called FSQP [68], iterating 1000 times with tight convergence requirements to find static bound width settings as close as possible to optimal.

Figure 3.4 shows the results of comparing the idealized version of our adaptive algorithm against the optimized static allocation, using a continuous AVG query over ten data sources under uniform and nonuniform costs. The x-axis shows the precision constraint δ, and the y-axis shows the overall cost per time unit. In a second experiment we used a workload of five AVG queries whose query sets were

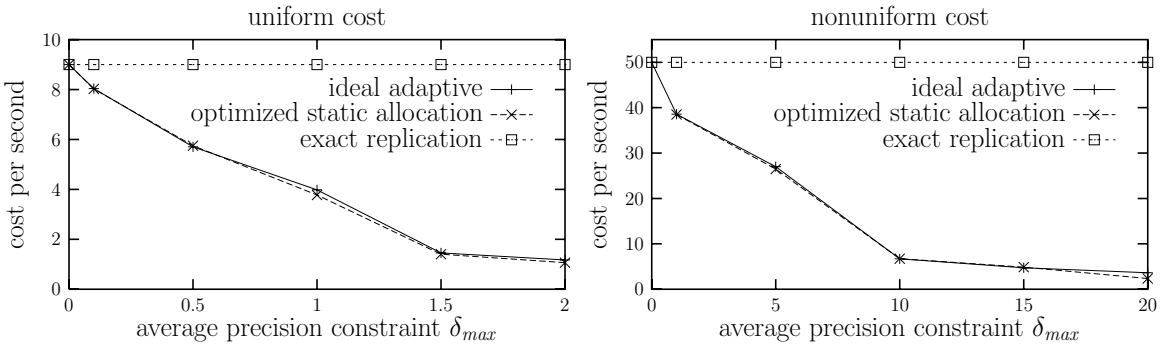


Figure 3.5: Ideal adaptive algorithm vs. optimized static allocation, multiple queries.

chosen randomly from the 10 objects. Figure 3.5 shows the result of this experiment for both uniform and nonuniform costs. The size of the query sets was assigned randomly between 2 and 5, and the precision constraint of each query was randomly assigned a value between 0 and δ_{max}, plotted on the x-axis. These results demonstrate that our adaptive bound width setting algorithm converges on bounds that are on par with those selected by an optimizer based on knowledge of the random walk step sizes.

3.4.6 Handling Precision Constraint Adjustments

Users may at any time choose to alter the precision constraint δ_j of any currently running continuous query Q_j. If the user increases δ_j (weaker precision), then additional bound width is allocated automatically by the bound growth algorithm at the central precision manager at the end of the current adjustment period. If the user decreases δ_j (stronger precision), bound growth is suppressed, and the automatic bound shrinking process will reduce the overall answer bound width over time until the requested precision level is reached. If an immediate improvement in answer precision is required, the central precision manager must proactively send messages to sources requesting explicitly that bounds be shrunk.

3.5 Coping with Latency

In a real implementation of our approach we must cope with message and computation latency. Suppose that each message, including refresh messages and bound growth messages, has an associated transmission latency as well as a processing delay by both the sender and receiver. We first note that due to such latencies, bound growth will be applied at sources after it is applied at the central data repository, and in the interim period the source copy of the bound is narrower than it could be. This phenomenon leads to a chance that some unnecessary updates are transmitted to the data repository, but correctness is not jeopardized. To reduce the delay for growth messages and lessen the chance of unnecessary refreshes, the data repository can begin the growth allocation process prior to the end of each adjustment period, and base the computations on preliminary refresh rate estimates.

Communication and computation latency for refresh messages is of more concern because, if handled naively, continuous queries may not access consistent data across all sources, leading to incorrect answers. To ensure continuous query answers based on consistent data, source timestamp all updates transmitted to the data repository. (We assume closely synchronized clocks, as in [67, 79].) Similarly, the precision manager timestamps all bound width updates with an adjustment period boundary. Value and width updates are converted into bound updates via the bound cache (recall Figure 3.2). Bound updates also have associated timestamps (we will discuss how they are assigned shortly), and our CQ evaluator (Figure 3.2) treats bound update timestamps as logical update times for the purposes of query processing. Correctness can only be guaranteed if the CQ evaluator receives bound updates monotonically in timestamp order, in which case it produces a new output value for every unique timestamp it receives as part of any update. When multiple updates have the same timestamp, the query evaluator treats them as a single atomic transaction and only produces a new output value for the last update with the same timestamp.

To ensure that the CQ evaluator receives bound updates that represent a consistent state and arrive in timestamp order, the bound cache in the central data repository is implemented using a combination of two *serializing queues* (described

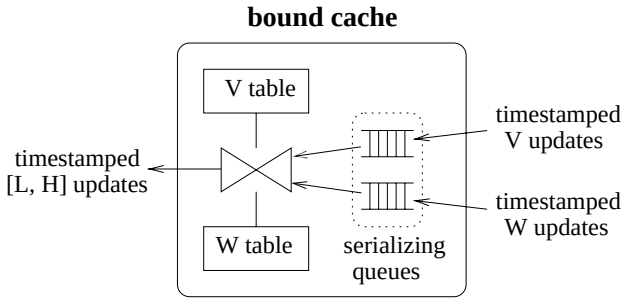


Figure 3.6: Bound cache for consistent and ordered bound updates.

shortly) and a *symmetric hash join* [115] (or other non-blocking join operator), as illustrated in Figure 3.6. The join operator combines value updates with width updates to produce bound updates that mirror the bounds maintained by sources, using object identifier equality as the join condition. Each hash table stores only the most recent value or width update for each object, based on timestamp, and each join result is assigned a timestamp equal to the timestamp of the input that generated the result.

Join inputs must arrive in timestamp order to ensure correct behavior. One way to guarantee global timestamp ordering across all V and W refresh streams is to delay processing of each update received on a particular stream until at least one update with a greater timestamp has been received on each of the other streams [67]. This approach is impractical in our setting, however, because it can result in unbounded delays unless additional communication is performed, and delays tend to be longer when the number of replicated objects is large. Instead, we take an approach similar to one taken in the field of streaming media to handle unordered packets with variable latency (see, *e.g.*, [82]), which relies on a reasonable latency upper bound. In our approach, serializing queues are positioned between the value and width refresh streams and the join. The effect of each serializing queue is to order updates by timestamp, and release each update U as soon as the current time t_{now} reaches $t_U + \lambda$, where t_U is U's timestamp and λ is the *latency tolerance*: an upper bound on the latency for any refresh message that holds with high probability and is determined empirically based on the networking environment. As long as all update messages obey this latency tolerance and appropriate queue scheduling is used, we can

be assured that the serializing queues together output to the join a monotonic stream of refreshes ordered by timestamp. Of course in practice occasional messages may be delayed by more than λ, resulting in temporary violations of precision guarantees, an unavoidable effect in any distributed environment with unbounded delays. Larger values of λ reduce the likelihood that update messages arrive late, but also increase the delay before results are released to the user. In Section 3.6.3 we show that using a reasonable choice of λ, late update messages are very rare.

3.5.1 Exploiting Constrained Change Rates

In some applications, certain data objects may have known maximum change rates, or at least bounds on change rate that hold with very high probability. If each data object O_i participating in a continuous query Q has maximum change rate R_i, then an approximate answer to Q that bounds the answer at time $t_{now} - \epsilon$ (for some local processing delay ϵ at the central data repository), rather than time $t_{now} - \lambda - \epsilon$, can be provided by having the data repository "pad" the bounds to account for recent changes rather than using serializing queues with a built-in delay as discussed above. Padding is performed by adding $\phi_i = 2 \cdot R_i \cdot \lambda$ symmetrically to the width of each updated bound $[L_i, H_i]$ after it is produced by the join. If this technique is employed, a reduced precision constraint $\delta'_Q \leq \delta_Q$ should be used for the purposes of bound width allocation and adjustment to ensure that padded answer bounds meet the original precision constraint δ_Q. The value of δ'_Q depends on the amount of padding and the type of query. For example, for AVG queries, we can set $\delta'_Q = \delta_Q - \frac{1}{|\mathcal{S}_Q|} \cdot \sum_{1 \leq i \leq n, O_i \in \mathcal{S}_Q} \phi_i$.

3.6 Implementation and Experimental Validation

We evaluated the performance of our technique and its practical applicability by building a real network traffic monitoring system. The system currently runs continuous queries over 10 hosts in our research group's network, following Example 2 from Section 3.2. In our implementation, a special monitoring program executes on each host.

It captures network traffic activity using the *TCPdump* utility and computes packet rate measurements as needed by the queries in the workload, representing them as time-varying numerical data objects. We use time units of one second, which matches the granularity at which our TCPdump monitor is able to capture data. Each host acts as a data source, and in all cases objects and their updates correspond to a one-minute moving window over packet rate measurements. We use queries $Q_1 - Q_5$ from Example 2 of Section 3.2, so different experiments use different objects. For example, query Q_5 uses one object for each of the 10 sources (hosts) for the overall windowed traffic volume between that host and external hosts. Each data object is assigned a bound width to be used for approximate replication. Bounds are cached at a central monitoring station, which updates the aggregated answers to continuous queries as bound widths shrink and grow and as data refreshes arrive. The communication cost (refresh or growth message) for each object is modeled as a uniform unit cost.

The first step in our experimentation was to determine good settings for the two algorithm parameters $\mathcal{T}$ (adjustment period) and S (shrink percentage). We experimented with a real-world network traffic data set in our simulator, with both uniform and nonuniform costs, and also with live data in our network monitoring implementation, and found that the following settings worked well in general: $\mathcal{T} = 10$ time units to achieve low growth message overhead relative to the timescale at which the data changes, and $S = 0.05$ (5%) to allow adaptivity while avoiding erratic bound width adjustments that tend to degrade performance. We also determined that our algorithm is not highly sensitive to the exact parameter settings. Setting or adjusting these parameters automatically is a topic of future work.

3.6.1 Single Query

We now present our first experimental results showing the effectiveness of our algorithm. We begin by considering a simple case involving a single continuous AVG query. We used query Q_5 from Example 2 of Section 3.2 applied over the 10 sources. Q_5 monitors the average rate of traffic to and from our organization, which ranged from about 100 to 800 packets per second.

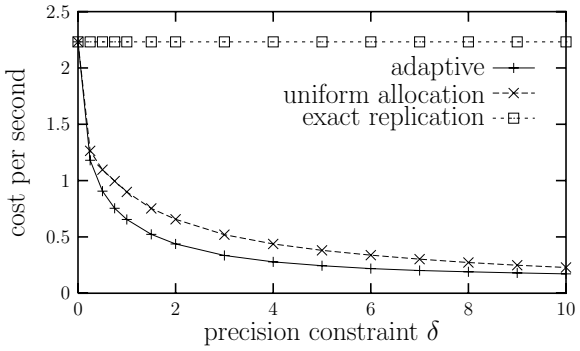


Figure 3.7: Adaptive algorithm vs. uniform static bound setting, query Q_5 using network monitoring implementation.

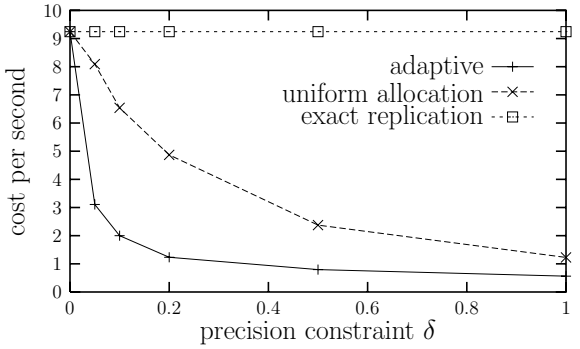


Figure 3.8: Adaptive algorithm vs. uniform static bound setting, single query over large-scale network data using simulator.

Since the optimized static bound width allocation described in Section 3.4.5 relies on knowing the random walk step size, it is not applicable to real-world data so cannot be used for comparison. Assuming data update patterns are not known in advance, the only obvious method of static allocation is to set all bound widths uniformly. Thus, we compare our algorithm against this setting.

Figure 3.7 compares the overall communication cost incurred in our real-world implementation by our adaptive algorithm compared to uniform static allocation, measuring cost for 21 hours after an initial warm-up period. The continuous query monitors the average traffic level with precision constraint δ ranging from 0 to 10 packets per second. Our algorithm offers a mild improvement over uniform bound

allocation for a single query, bearing in mind that the experiment was over small-scale network monitoring data available for monitoring on a few hosts within our organization.

To test our algorithm on large-scale network data with many hosts, we ran a simulation on publicly available traces of network traffic levels between hosts distributed over a wide area during a two hour period [87]. For each host, average packet rates ranged from 0 to about 150 packets per second, and we randomly selected 200 hosts as our simulated data sources. Figure 3.8 shows the results using our simulator over this large-scale data set, accounting for all communication costs. With this data set our algorithm significantly outperforms uniform static allocation for queries that can tolerate a moderate level of imprecision (small to medium precision constraints). For queries with very weak precision requirements (large precision constraints), even naive allocation schemes achieve low cost, and the slight additional overhead of our algorithm causes it to perform about on par with uniform static allocation.

3.6.2 Multiple Queries

We now describe our experiments with multiple continuous queries having overlapping query sets. We used a workload of the five continuous AVG queries $Q_1 - Q_5$ from Example 2 in Section 3.2. $2^5 - 1$ "measurement groups" are defined at each source based on which subsets of the five query predicates a packet satisfies. Each measurement group is aggregated and acts as a data object replicated (approximately) at the central monitoring station. (It may seem more natural for sources to further aggregate data objects into one object per query; we discuss this option shortly.)

Figure 3.9 shows the results of our experiments measuring cost for 23 hours after an initial warm-up period. The x-axis shows the precision constraints used for queries Q_2 and Q_5. The other queries monitored a much lower volume of data (by a factor of roughly 100) so for each run we set their precision constraints to 1/100th that shown on the x-axis. As discussed in Section 3.3.1, uniform static bound width allocation can be performed for multiple overlapping queries if for each data object involved in more than one query we maintain the narrowest bound assigned. Our algorithm

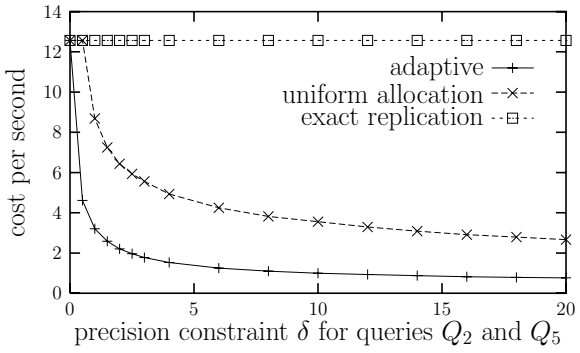


Figure 3.9: Adaptive algorithm vs. uniform static bound setting, queries $Q_1 - Q_5$ using network monitoring implementation.

significantly outperforms uniform static allocation for queries that can tolerate a moderate level of imprecision (small to medium precision constraints). For example, using reasonable precision constraints of $\delta = 4$ for queries Q_2 and Q_5 and $\delta = 0.04$ for queries Q_1, Q_3, and Q_4, our algorithm achieves a cost of only 1.6 messages per second, compared with a cost of 5.4 with uniform static bound width allocation. Furthermore, as with all previous results reported, the overall cost decreases rapidly as the precision constraint is relaxed, offering significant reductions in communication cost compared with performing a refresh on every update (labeled "exact replication").

Source Aggregation

In the multiple-query workload it may appear advantageous for sources to further aggregate data objects to form one object per query whose refreshes are sent to the monitoring station, instead of one per query subset. Interestingly, doing so (a process we call *source aggregation*) does not always result in lower overall cost, and whether it is cheaper to perform source aggregation depends on the data, query workload, and user-specified precision constraints.

We analyze the effect of source aggregation mathematically, and show that cases exist where source aggregation is advantageous in terms of minimizing communication cost, and cases also exist where it is not. Suppose there are three data values $V(O_0)$, $V(O_1)$, and $V(O_2)$ at a single data source, and two continuous SUM queries Q_1 and Q_2 at the data repository. Q_1 computes $V(O_0)+V(O_1)$ with a precision constraint of

δ_1, and Q_2 computes $V(O_0)+V(O_2)$ with precision constraint δ_2. If source aggregation is not applied, then three bounds are maintained by the data repository: $[L_0, H_0]$, $[L_1, H_1]$, and $[L_2, H_2]$, corresponding to the three source values $V(O_0)$, $V(O_1)$, and $V(O_2)$. Let width $W_i = H_i - L_i$. The query precision constraints require that $W_0 + W_1 \leq \delta_1$ and $W_0 + W_2 \leq \delta_2$. On the other hand, if source aggregation is applied, then two bounds are maintained by the data repository: a bound $[L_{Q1}, H_{Q1}]$ on $V_{Q1} = V(O_0) + V(O_1)$ and a bound $[L_{Q2}, H_{Q2}]$ on $V_{Q2} = V(O_0) + V(O_2)$, where $W_{Q1} \leq \delta_1$ and $W_{Q2} \leq \delta_2$.

If all refresh costs are equal, then cost is determined by the sum of the refresh frequencies of all bounds, which can be estimated using our random walk model. Recall from Section 3.4.3 that the refresh frequency for a bound of width W_i on the value $V(O_i)$ is $F_i \approx \frac{(2 \cdot s_i)^2}{(W_i)^2}$, where s_i is the random walk step size of $V(O_i)$. Applying the law that variances are additive for sums of independent random variables [44], we can compute the refresh frequency for the source-aggregated bounds $F_{Q1} \approx \frac{((2 \cdot s_0)^2 + (s \cdot s_1)^2)}{(W_{Q1})^2}$ and $F_{Q2} \approx \frac{((2 \cdot s_0)^2 + (s \cdot s_2)^2)}{(W_{Q2})^2}$. Therefore, the overall communication cost if source aggregation is not applied is $\mathcal{C}_n = F_0 + F_1 + F_2$, and the overall cost if source aggregation is applied is $\mathcal{C}_p = F_{Q1} + F_{Q2}$.

Suppose $\delta_1 \gg \delta_2$, so there is a large disparity in the precision constraints of the two queries. For this purpose we can treat δ_1 as ∞, and it is easy to verify mathematically that for any step sizes and any bound widths that meet the constraints, $\mathcal{C}_p \leq \mathcal{C}_n$ so source aggregation achieves lower overall communication cost. On the other hand, suppose $s_0 \gg s_1$ and $s_0 \gg s_2$, so value $V(O_0)$ involved in both queries changes much more rapidly than $V(O_1)$ and $V(O_2)$, which are each only involved in a single query. If we take the extreme case where $s_1 = s_2 = 0$, then if source aggregation is not performed any reasonable width allocation strategy will assign $W_1 = 0$, $W_2 = 0$, and $W_0 = \min\{\delta_1, \delta_2\}$, and we can derive that $\mathcal{C}_n \leq \mathcal{C}_p$.

Conceptually, if there is a significant disparity between the precision constraints of overlapping queries, source aggregation achieves lower overall communication cost for refresh message transmission because queries with large precision constraints can use separate wide bounds not constrained by other queries with small precision constraints. On the other hand, if most updates are to objects involved in multiple

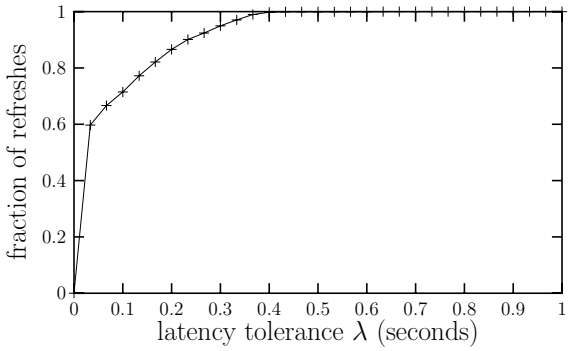


Figure 3.10: Fraction of refreshes arriving after the maximum latency tolerance λ for query Q_5.

queries, it is preferable in terms of overall communication cost not to apply source aggregation, to avoid redundantly applying those updates to one object per relevant query. As an extreme case, consider Example 1 from Section 3.2 in which each source (router) maintains a single queue latency value accessed by multiple path latency queries, and all updates at each source apply to objects involved multiple queries. Source aggregation would have each router maintain one copy of its queue size measurement for each path latency query, each with a bound having a potentially different width. Updates would fall outside the bounds at different times causing unnecessary updates to be transmitted to the central data repository.

As future work we plan to design and experiment with an algorithm that monitors the expected cost of using versus not using source aggregation and switches adaptively between them. Note that the choice of whether to perform source aggregation can be made independently for each source and for each independent set of overlapping queries, and the best overall configuration may be to perform source aggregation selectively.

3.6.3 Impact of Message Latency

Our last experiment measures refresh message latency. In Figure 3.10 we vary the maximum latency tolerance λ (recall Section 3.5) and measure the fraction of refreshes arriving within λ for query Q_5 during a 21-hour period. In our implementation

refreshes are transmitted over a local area network. A value of $\lambda = 0.4$ seconds, which is reasonable since data changes are meaningful on a scale of about 1 second in our case, ensures that 99.8% of refreshes are received on time. When a moderate precision constraint for this query of $\delta = 5$ is used, refreshes exceeding the latency allowance occur only about once every 65.7 minutes. When a refresh does arrive late, the resulting inconsistency in the output is brief, and based on our measurements plotted in Figure 3.10 the overall fraction of time the answer is consistent (*fidelity* in the terminology of [100]) is at least 99.997%. By adjusting λ, higher fidelity can be achieved at the expensive of delayed output, or vice-versa. ([82] proposes an algorithm for adjusting the latency tolerance adaptively in a similar context based on observed latency distributions.)

3.7 Related Work

We provided an overview of related work in Section 1.5. Here we elaborate on some of the most closely related work and discuss other work specifically related to Chapters 3 through 5, which deal with bounded value approximations.

There is a large body of work dedicated to systems that improve query performance by giving approximate answers. Early work in this area is reported in [83]. Most of these systems use either precomputation (*e.g.*, [90]), sampling (*e.g.*, [50]), or both (*e.g.*, [42]) to give an answer with statistically estimated bounds, without scanning all of the input data. By contrast, in our work, all of the data is accessed (usually indirectly by reading approximate replicas), to provide guaranteed rather than statistical results.

Data objects whose values are interval bounds can be considered a special case of constrained values in *Constraint Databases* [95, 60, 18, 19, 64, 13], or as null variables with local conditions in *Incomplete Information Databases* [1]. However, no work in these areas that we know of considers constrained values as bounded approximations of exact values stored elsewhere.

In the *multi-resolution relational data model* [94], data objects undergo various degrees of lossy compression to reduce the size of their representation. By reading the

compressed versions of data objects instead of the full versions, the system can quickly produce approximate answers to queries. By contrast, in our work performance is improved by reducing the frequency with which data objects read from remote sources, rather than by reducing the size of the data representation.

Another body of work that deals with imprecision in information systems is *Information Quality* (*IQ*) research, *e.g.*, [85]. IQ systems quantify the accuracy of data at the granularity of an entire data server. Since no bounds are placed on individual data values, queries have no concrete knowledge about the precision of individual data values from which to form a bounded answer. Therefore, IQ systems cannot give a guaranteed bound on the answer to a particular query.

A much more closely related body of research stems from work on *quasi copies* [7]. With quasi copies, centrally maintained approximations are permitted to deviate from exact source values by constrained amounts, thereby providing precision guarantees. Recent work [100] extends the ideas of [7], proposing an architecture in which a network of repositories cooperate to deliver data with precision guarantees to a large population of remote users. In an environment with multiple cooperating repositories, data may be propagated through several nodes before ultimately reaching the end-user application, so latency can be a significant concern. The work in [100] focuses on selecting topologies and policies for cooperating repositories to minimize the degree to which latency causes precision guarantees to be violated. However, the work in [7] and [100] does not address queries over multiple data values, whose answer precision is a function of the precision of the input values, so they do not need to deal with the optimization problems we address for minimizing communication.

Recent work on reactive network monitoring [31] addresses scenarios where users wish to be notified whenever the sum of a set of values from distributed sources exceeds a prespecified critical value. In their solution each source notifies a central processor whenever its value exceeds a certain threshold, which can be either a fixed constant or a value that increases linearly over time. The local thresholds are set to guarantee that in the absence of notifications, the central processor knows that the sum of the source values is less than the critical value. The thresholds in [31], which are related to the bounds in our algorithm, are set uniformly across all sources.

Similarly, in [125], which focuses on bounded approximate values under symmetric replication, error bounds are allocated uniformly across all sites that can perform updates. In contrast to these approaches, we propose a technique in which bound widths are allocated nonuniformly and adjusted adaptively based on refresh costs and data change rates.

Maintaining numeric bounds on aggregated values from multiple sources can be thought of as ensuring the continual validity of distributed constraints. Most work on distributed constraint checking, *e.g.*, [14, 46, 56] only considers insertions and deletions from sets, not updates to data values. We are aware of three proposals in which sources communicate among themselves to verify numerical consistency constraints across sources containing changing values: *data-value partitioning* [102], the *demarcation protocol* [11], and recent work by Yamashita [123]. The approach of these proposals could in principle be applied to our setting: sources could renegotiate bound widths in a peer-to-peer manner with the goal of reducing refresh rates. However, in many scenarios it may be impractical for sources to keep track of the other sources involved in a continuous query (or many continuous queries) and communicate with them directly, and even if practical it may be necessary to contact multiple peers before finding one with adequate "spare" bound width to share. It seems unlikely that the overhead of inter-source communication is warranted to potentially save a single refresh message. Furthermore, the algorithms in [11, 102, 123] are not designed for the purpose of minimizing communication cost, and they do not accommodate multiple queries with overlapping query sets.

Some work on real-time databases, *e.g.*, [72], focuses on scheduling multiple complex, time-consuming computation tasks that yield imprecise results that improve over time. In contrast, our work does not focus on how best to schedule computations, but rather on how to reduce refresh rates for replicas in a distributed environment while bounding the resulting imprecision.

3.8 Chapter Summary

In this chapter we specified one of the pieces of our overall approach achieving efficient communication resource utilization in an environment of centralized query processing over distributed data sources. We focused on continuous queries with custom precision constraints, which are used to avoid refreshing replicas each time an update to the master copy occurs while still guaranteeing sufficient precision of query results at all times. Users or applications may trade precision for lower communication cost at a fine granularity by individually adjusting precision constraints of continuous queries. Imprecision of query results is bounded numerically so applications need not deal with any uncertainty.

To validate the techniques proposed in this chapter we performed a number of experiments using simulations and a real network monitoring implementation. Our experiments demonstrated:

- For a steady-state scenario our algorithm converges on bound widths that perform on par with those selected statically using an optimization problem solver with complete knowledge of data update behavior.
- In the case of a single continuous query, our algorithm significantly outperforms uniform bound width allocation in some cases, and in other cases our algorithm is only somewhat better than uniform allocation. As future work we plan to characterize those cases for which our algorithm achieves a significant improvement over uniform static allocation, and those cases for which uniform allocation suffices.
- In the case of multiple overlapping continuous queries, our algorithm significantly outperforms uniform bound width allocation.

Chapter 4

Answering Unexpected One-Time Queries

4.1 Introduction

Providing guaranteed precision for a workload of continuous queries, as addressed in Chapter 3, is not sufficient to handle an important class of queries that arises frequently in practice: one-time, impromptu queries. Users or applications interacting with the data repository may at any time desire to obtain a *one-time* result of a certain query, which includes a precision constraint and may or may not be the same as one of the continuous queries currently being evaluated. In many cases one-time queries are issued in response to results observed via currently executing continuous queries, in order to obtain more detailed information in two ways: (1) one-time queries are likely to request improved precision over the corresponding continuous queries, and (2) they may target specific portions of the data being monitored.

For case (1), when an ad-hoc one-time query is issued the data replicas in the repository may not be of sufficient precision to meet the query's precision constraint. To obtain a query answer of adequate precision it may be necessary to access master copies of a subset of the queried data objects by contacting remote sources, incurring performance penalties. Case (2) suggests that one-time queries may include selection predicates over approximately replicated values to narrow the scope of the query. In

this chapter we study the problem of maximizing communication performance in the presence of unanticipated one-time aggregation queries with precision constraints and optional selection predicates. To solve this problem we devise efficient algorithms for minimizing accesses to remote master copies.

4.1.1 Chapter Outline

The remainder of Section 4.1 is devoted to an example scenario that we use to motivate our approach and refer to throughout this chapter for examples. In Section 4.2 we discuss the execution of aggregation queries with optional selection predicates over approximate replicas. We present our specific optimization algorithms for aggregation queries over sets of objects in Sections 4.3 and 4.4. In Section 4.5 we present some preliminary results for aggregation queries with joins. Finally we discuss related work specific to this chapter in Section 4.6 and summarize this chapter briefly in Section 4.7.

4.1.2 Running Example

As a scenario for motivation and examples throughout this chapter, we will again consider a replication system used for monitoring a computer network. In this example, instead of analyzing actual packets as in our previous example (Section 3.1.3), we assume the system computes measurements and makes them available. Each node (computer) in the network tracks the average latency, bandwidth, and traffic level for each incoming network link from another node. Administrators at a central monitoring station analyze the status of the network by collecting data periodically from the network nodes. For each link $N_i \rightarrow N_j$ in the network, the monitoring station will maintain replicas of the latest latency, bandwidth, and traffic level figures obtained from node N_j. A certain number of continuous queries with precision constraints may be executing over approximate replicas, with precision levels set and adjusted as discussed in Chapter 3. However, administrators may also wish to issue ad-hoc, one-time queries over the replicated data, such as:

Q_1 What is the bottleneck (minimum bandwidth link) along a path $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_k$?

Q_2 What is the total latency along a path $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_k$?

Q_3 What is the average traffic level in the network?

Q_4 What is the minimum traffic level for fast links (*i.e.*, links with high bandwidth and low latency)?

Q_5 How many links have high latency?

Q_6 What is the average latency for links with high traffic?

Collecting new data values from each relevant node every time a one-time query is posed to compute exact answers would take too long and might adversely affect the system. Requiring that all nodes constantly send their updated values to the monitors is also expensive and generally unnecessary, as discussed in Section 3.2. Since administrators do not usually require exact precision for every query, we permit a numeric precision constraint to be submitted along with each one-time query, just as with continuous queries, indicating how wide a bound is tolerable in the one-time answer. For example, suppose the administrator wishes to obtain the latency in some area in response to a complaint about high latency from a customer. To determine whether the latency experienced by the customer is occurring within the region of control of the administrator, the administrator does not need to know the precise latency, but may wish to obtain an answer to within 5 milliseconds of precision.

The small table in Figure 4.1 shows sample data replicated in the form of numeric intervals at a central network monitoring station (acting as the data repository), along with the current exact values at the network nodes (acting as data sources). The *weights* may be ignored for now. Each row in Figure 4.1 corresponds to a network link between the *link from* node and the *link to* node. Recall that exact master values for *latency*, *bandwidth*, and *traffic* for incoming links are measured and stored at the *link to* node. In addition, for each link, the monitoring station stores a bound for *latency*, *bandwidth*, and *traffic*, which serve as approximate replicas of the exact values. The monitoring station can use these bounds to compute bounded answers to one-time queries, just as for continuous queries (Chapter 3).

Suppose a bounded answer to a query with aggregation is computed from approximate replicas, but the answer does not satisfy the user's precision constraint, *i.e.*,

	link		*latency*		*bandwidth*		*traffic*		*refr.*	*weights*		
	from	*to*	*replica*	*exact*	*replica*	*exact*	*replica*	*exact*	*cost*	W	W'	W''
1	N_1	N_2	[2, 4]	3	[60, 70]	61	[95, 105]	98	3	2	10	29.5
2	N_2	N_4	[5, 7]	7	[45, 60]	53	[110, 120]	116	6	2	10	2
3	N_3	N_4	[12, 16]	13	[55, 70]	62	[95, 110]	105	6		15	41.5
4	N_2	N_3	[9, 11]	9	[65, 70]	68	[120, 145]	127	8		25	2
5	N_4	N_5	[8, 11]	11	[40, 55]	50	[90, 110]	95	4	3	20	36.5
6	N_5	N_6	[4, 6]	5	[45, 60]	45	[90, 105]	103	2	2	15	31.5

Figure 4.1: Sample data for network monitoring example.

the answer bound is too wide. This scenario may occur, for example, when bound widths are set according to currently running continuous queries with weak precision requirements. In this case, some data must be accessed from sources to improve precision. Recall that we assume that there is a known quantitative cost associated with refreshing data objects from their sources, and this cost may vary for each data item (*e.g.*, in our example it might be based on the node distance or network path latency). We show sample refresh costs for our example in Figure 4.1. Our system uses optimization algorithms that attempt to find the best combination of replicated bounds and master values to use in answering a query, in order to minimize the cost of remote refreshes while still guaranteeing the precision constraint.

Next we discuss the execution of aggregation queries with optional selection predicates over approximate replicas, before presenting our specific optimization algorithms for aggregation queries over sets of objects in Sections 4.3 and 4.4.

4.2 Execution of One-Time Queries

Executing an aggregation query with a precision constraint may involve combining exact data stored on remote sources with approximate replicas stored in a local repository. In this section we describe in general how one-time queries are executed, and we present a cost model to be used by our algorithms that choose remote replicas to access when answering queries. When a replica at a remote source is accessed by the repository to improve the precision of a one-time query, the source responds by sending a refresh message back to the repository. A refresh message for object O

consists of O's current exact value $V(O)$ and a new bound $[L_O, H_O]$ that contains $V(O)$.

In this chapter we consider selection predicates, and to simplify discussion of them we assume the relational model until the end of this chapter. Our techniques of this chapter and the entire dissertation can be implemented with any data model that supports aggregation of numerical values. We assume that replication may be performed at the granularity of individual *tuples*, which can be thought of as the data objects of the relational model. For now we consider single-table aggregation queries of the following form. Joins are addressed in Section 4.5.

SELECT `AGGREGATE`(T.a) WITHIN δ
FROM T
WHERE `PREDICATE`

`AGGREGATE` is one of the standard relational aggregation functions: COUNT, MIN, MAX, SUM, or AVG. `PREDICATE` is any predicate involving columns of table T and possibly constants. δ is a nonnegative real constant specifying the precision constraint, which requires that the bounded answer $[L, H]$ to the query satisfies $0 \leq H - L \leq \delta$. If δ is omitted then $\delta = 0$ implicitly. Note that this language for one-time querying allows slightly more general queries than those studied in the CQ context in Chapter 3. In particular, selection predicates over approximate replicas are permitted.

4.2.1 Computing Bounded Answers

To compute a bounded answer to a query of this form, our system executes several steps:

1. Compute an initial bounded answer based on the current replicated bounds and determine if the precision constraint is met. If not:

2. An algorithm CHOOSE_REFRESH examines the repository's copy of table T and chooses a subset of T's tuples T_R to refresh. The source for each tuple in T_R is asked to refresh the repository's copy of that tuple.

3. Once the refreshes are complete, recompute the bounded answer based on the repository's now partially updated copy of T.

Our CHOOSE_REFRESH algorithm ensures that the answer after step 3 is guaranteed to satisfy the precision constraint.

4.2.2 Answer Consistency

A simple *multiversion concurrency control scheme* [15] can be used to ensure that the answer computed in step 3 is based on a consistent snapshot of all the data.[1] First, in step 1, updates to the data repository's bound cache (Figure 3.6) are halted momentarily while exact values for all bounds relevant to the query are accessed by requesting refreshes and the timestamp $\mathcal{T}$ of the most recently applied update is read. A temporary copy of all bounds read by the query is made at the data repository along with $\mathcal{T}$, which represents the time as of which the one-time query is evaluated. $\mathcal{T}$ is included with all refresh requests in step 2, and sources respond with a refresh message that includes the exact value as of that time, thereby ensuring that the answer computed in step 3 will represent a consistent snapshot as of time $\mathcal{T}$. (Recall from Section 3.5 that we assume that all nodes maintain closely synchronized clocks.) The new bounds included in the refresh messages are selected based on the most recent exact source values, and upon receipt at the repository they are placed in the serializing queues for later insertion into the bound cache as described in Section 3.5.

To enable access to recent historical data for query answer consistency purposes, sources must maintain a brief version history for each data value. Each time a source receives an access request with timestamp $\mathcal{T}$ it may discard all versions with timestamps earlier than $\mathcal{T}$. (If the storage requirements of multiple versions is of concern, it may be appropriate to use more aggressive policies for determining that old versions can be discarded safely, although additional communication may be necessary.)

[1] In contrast, multiversion concurrency control is not required to ensure consistent answers for continuous queries because in our approach (Chapter 3) CQ answers are computed entirely from approximate replicas, and are not formed by combining approximate replicas with remote source data as in our approach to answering one-time queries.

The above consistency scheme is appropriate for data captured from real-time measurements such as sensor signals or network traces, which have associated real-world timestamps. In contrast, with *transactional data* such as stock market trades or auction bids, in which the writes to a database define the real-world outcome, exact timestamps are not typically of primary importance. For transactional data a more appropriate consistency criterion is *one-copy serializability* [15]. In-depth treatment of transactions and serializability for approximately replicated data is beyond the scope of this dissertation. However, we point out that well-known *distributed concurrency control* protocols [15] can be used to enforce one-copy serializability over bounds replicated between sources and the central repository. Communication is only necessary when master values exceed their bounds, and it is ensured that each query's answer bound reflects the state of the data at a transaction boundary, thereby avoiding approximate answers derived from transactionally inconsistent data.

4.2.3 Overview

The remainder of this chapter presents the details of steps 1-3 in Section 4.2.1. In particular, for each type of aggregation query we address the following two problems:

- How to compute a bounded answer based on the current replicated bounds. This problem corresponds to steps 1 and 3 above.

- How to choose the set of tuples to refresh. This problem corresponds to step 2 above. A CHOOSE_REFRESH algorithm is *optimal* if it finds the cheapest subset T_R of T's tuples to refresh (*i.e.*, the subset with the least total cost) that guarantees the final answer to the query will satisfy the precision constraint for any exact values of the refreshed tuples within the current bounds.

We are assuming that the cost to refresh a set of tuples is the sum of the costs of refreshing each member of the set, in order to keep the optimization problem manageable. This simplification ignores possible amortization due to batching multiple requests to the same source. Also recall that we assume a separate refresh cost may

be assigned to each tuple, although in practice all tuples from the same source may incur the same cost.

Note that the entire set T_R of tuples to refresh is selected before the refreshes actually occur, so the precision constraint must be guaranteed for any possible exact values for the tuples in T_R. A different approach is to refresh tuples one at a time (or one source at a time), computing a bounded answer after each refresh and stopping when the answer is precise enough. See Chapter 7 (Section 7.1.2) for further discussion of this alternative.

Sections 4.3 and 4.4 present details based on each specific aggregation function, considering queries with and without selection predicates, respectively. Section 4.5 briefly discusses joins.

4.3 Aggregation without Selection Predicates

This section specifies how to compute a bounded answer from bounded data values for each type of aggregation function, and describes algorithms for selecting refresh sets for each aggregation function. For now, we assume that any selection predicate in the query involves only columns that contain exact values. Thus, in this section we assume that the selection predicate has already been applied and the aggregation is to be computed over the tuples that satisfy the predicate. Queries with selection predicates involving columns that contain approximate replicas are covered in Section 4.4, and joins involving approximate replicas are discussed in Section 4.5.

Suppose we want to compute an aggregate over column $T.a$ of a replicated table T. The value of $T.a$ for each tuple t_i is stored in the repository as a bound $[L_i, H_i]$. While computing the aggregate, the query processor has the option for each tuple t_i of either reading the replicated bound $[L_i, H_i]$ or refreshing t_i to obtain the master value $V(O_i)$. The cost to refresh t_i is C_i. The final answer to the aggregate is a bound $[L_A, H_A]$.

4.3.1 Computing MIN with No Selection Predicate

Computing the bounded MIN of $T.a$ is straightforward:

$$[L_A, H_A] = [\min_{t_i \in T}(L_i), \min_{t_i \in T}(H_i)]^2$$

The lowest possible value for the minimum (L_A) occurs if for all $t_i \in T$, $V(O_i) = L_i$, *i.e.*, each value is at the bottom of its bound. Conversely, the highest possible value for the minimum (H_A) occurs if $V(O_i) = H_i$ for all tuples. Returning to our example of Section 4.1.2, suppose we want to find the minimum bandwidth link along the path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$, *i.e.*, query Q_1. Applying the bounded MIN of *bandwidth* to tuples $T = \{1, 2, 5, 6\}$ in Figure 4.1 yields $[40, 55]$.

Choosing an optimal set of tuples to refresh for a MIN query with a precision constraint is also straightforward, although the algorithm's justification and proof of optimality is nontrivial. The CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$ algorithm chooses T_R to be all tuples $t_i \in T$ such that $L_i < \min_{t_k \in T}(H_k) - \delta$, where δ is the precision constraint, independent of refresh cost. That is, T_R contains all tuples whose lower bound is less than the minimum upper bound minus the precision constraint. If B-tree indexes exist on both the upper and lower bounds, the set T_R can be found in time less than $O(|T|)$ by first using the index on upper bounds to find $\min_{t_k \in T}(H_k)$, and then using the index on lower bounds to find tuples that satisfy $L_i < \min_{t_k \in T}(H_k) - \delta$. Without these two indexes, the running time for CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$ is $O(|T|)$.

Consider again our example query Q_1, which finds the minimum bandwidth along path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$. CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$ with $\delta = 10$ would choose to refresh tuple 5, since it is the only tuple among $\{1, 2, 5, 6\}$ whose low value is less than $\min_{t_k \in \{1,2,5,6\}}(H_k) - \delta = 55 - 10 = 45$. After refreshing, tuple 5's bandwidth value turns out to be 50, so the new bounded answer is [45, 50].

[2] In this and all subsequent formulas, we define $\min(\emptyset) = +\infty$ and $\max(\emptyset) = -\infty$.

Proof of Correctness of CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$

The CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$ algorithm chooses T_R to be all tuples $t_i \in T$ such that $L_i < \min_{t_k \in T}(H_k) - \delta$, where δ is the precision constraint. To show that this choice for T_R is correct and optimal, we show that every tuple in T_R must appear in every solution, and that this solution is sufficient to guarantee the precision constraint. First, we show that every tuple in T_R must appear in every solution. Consider any $t_i \in T_R$ and suppose we choose to refresh every tuple in T except t_i. It is possible that refreshing all other tuples t_j results in $V(O_j) = H_j$ for each one (*i.e.*, each precise value is at its upper bound). In this case, after refreshing, our new bounded answer will be $[L_A, H_A]$ where $L_A \leq L_i$ and $H_A = \min_{t_k \in T}(H_k)$. Since $L_i < \min_{t_k \in T}(H_k) - \delta$ by the definition of T_R, $\min_{t_k \in T}(H_k) - L_i > \delta$, so $H_A - L_A > \delta$, and the precision constraint does not hold. Thus, every tuple in T_R must be in any solution to guarantee that the precision constraint will hold.

Next, we show that T_R is sufficient to guarantee the precision constraint. Let L_p be $\min_{t_k \in \overline{T_R}}(L_k)$, where $\overline{T_R} = T - T_R$. Note that for all $t_i \in \overline{T_R}$, L_i is within δ of $\min_{t_k \in T}(H_k)$, so we have $\min_{t_k \in T}(H_k) - L_p \leq \delta$. After tuples in T_R have been refreshed and current exact values are available at the repository, $\min_{t_k \in T}(H_k)$ can only decrease, so we know $H_A - L_p \leq \delta$. After refreshing the tuples in T_R, they will have a bound width of zero, *i.e.*, $L_i = H_i = V(O_i)$. There are thus two cases that can occur after the tuples $t_i \in T_R$ have been refreshed. First, if any of the values $V(O_i)$ are less than or equal to L_p, then we can compute an exact minimum. Otherwise, if all of the values $V(O_i)$ are greater than L_p, then $L_A = L_p$. Since $H_A - L_p \leq \delta$, it follows that $H_A - L_A \leq \delta$.

4.3.2 Computing MAX with No Selection Predicate

The MAX aggregation function is symmetric to MIN. Thus:

$$[L_A, H_A] = [\max_{t_i \in T}(L_i), \max_{t_i \in T}(H_i)]$$

and the CHOOSE_REFRESH$_{\text{NO_SEL/MAX}}$ algorithm chooses T_R to be all tuples $t_i \in T$ such that $H_i > \max_{t_k \in T}(L_k) + \delta$.

4.3.3 Computing SUM with No Selection Predicate

To compute the bounded SUM aggregate, we take the sum of the values at each extreme:

$$[L_A, H_A] = [\sum_{t_i \in T} L_i, \sum_{t_i \in T} H_i]$$

The smallest possible sum occurs when all values are as low as possible, and the largest possible sum occurs when all values are as high as possible. In our running example, the bounded SUM of *latency* along the path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$ (query Q_2 using the data from Figure 4.1 is [19, 28].

The problem of selecting an optimal set T_R of tuples to refresh for SUM queries with precision constraints is better attacked as the equivalent problem of selecting the tuples not to refresh: $\overline{T_R} = T - T_R$. We first observe that $H_A - L_A = \sum_{t_i \in T} H_i - \sum_{t_i \in T} L_i = \sum_{t_i \in T}(H_i - L_i)$. After refreshing all tuples $t_j \in T_R$, we have $H_j - L_j = 0$, so these values contribute nothing to the bound. Thus, after refresh, $\sum_{t_i \in T}(H_i - L_i) = \sum_{t_i \in \overline{T_R}}(H_i - L_i)$. These equalities combined with the precision constraint $H_A - L_A \leq \delta$ give us the constraint $\sum_{t_i \in \overline{T_R}}(H_i - L_i) \leq \delta$. The optimization objective is to satisfy this constraint while minimizing the total cost of the tuples in T_R. Observe that minimizing the total cost of the tuples in T_R is equivalent to maximizing the total cost of the tuples not in T_R. Therefore, the optimization problem can be formulated as choosing $\overline{T_R}$ so as to maximize $\sum_{t_i \in \overline{T_R}} C_i$ under the constraint $\sum_{t_i \in \overline{T_R}}(H_i - L_i) \leq \delta$.

It turns out that this problem is isomorphic to the well-known *0/1 Knapsack Problem* [28], which can be stated as follows: We are given a set S of items that each have weight W_i and profit P_i, along with a knapsack with capacity M (*i.e.*, it can hold any set of items as long as their total weight is at most M). The goal of the Knapsack Problem is to choose a subset S_K of the items in S to place in the knapsack that maximizes total profit without exceeding the knapsack's capacity. In other words, choose S_K so as to maximize $\sum_{i \in S_K} P_i$ under the constraint $\sum_{i \in S_K} W_i \leq M$. To state the problem of selecting refresh tuples for bounded SUM queries as the 0/1 Knapsack

Problem, we assign $S = T$, $S_K = \overline{T_R}$, $P_i = C_i$, $W_i = (H_i - L_i)$, and $M = \delta$.

Unfortunately, the 0/1 Knapsack Problem is known to be NP-Complete [41]. Hence all known approaches to solving the problem optimally, such as dynamic programming, have a worst-case exponential running time. Fortunately, an approximation algorithm exists that, in polynomial time, finds a solution having total profit that is within a fraction ϵ of optimal for any $0 < \epsilon < 1$ [57]. The running time of the algorithm is $O(n \cdot \log n) + O((\frac{3}{\epsilon})^2 \cdot n)$. We use this algorithm for CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$. Adjusting parameter ϵ in the algorithm allows us to trade off the running time of the algorithm against the quality of the solution.

In the special case of uniform costs ($C_i = C_j$ for all tuples t_i and t_j), all knapsack objects have the same profit P_i, and the 0/1 Knapsack Problem has a polynomial algorithm [28]. The optimal answer then can be found by "placing objects in the knapsack" in order of increasing weight W_i until the knapsack cannot hold any more objects. That is, we add tuples to $\overline{T_R}$ starting with the smallest $H_i - L_i$ bounds until the next tuple would cause $\sum_{t_i \in \overline{T_R}}(H_i - L_i) > \delta$. If an index exists on the bound width $H_i - L_i$ (see Section 7.1.2), this algorithm can run in sublinear time. Without an index on bound width, the running time of this algorithm is $O(n \cdot \log n)$, where $n = |T|$.

Consider again query Q_2 that asks for the total latency along path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$. Figure 4.1 shows the correspondence between our problem and the Knapsack Problem by specifying the knapsack "weight" $W = H - L$ for the *latency* column of each tuple in {1, 2, 5, 6}. Using the exponential (optimal) knapsack algorithm to find the total latency along path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$ with $\delta = 5$, tuples 2 and 5 are "placed in the knapsack" (whose capacity is 5), leaving T_R = {1, 6}. The bounded SUM of *latency* after refreshing tuples 1 and 6 is [21, 26].

Performance Experiments

CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ uses the approximation algorithm from [57] to quickly find a cheap set of tuples T_R to refresh such that the precision constraint is guaranteed to hold. We implemented the algorithm and ran experiments using 90 actual stock prices that varied highly in one day. The high and low values for the day

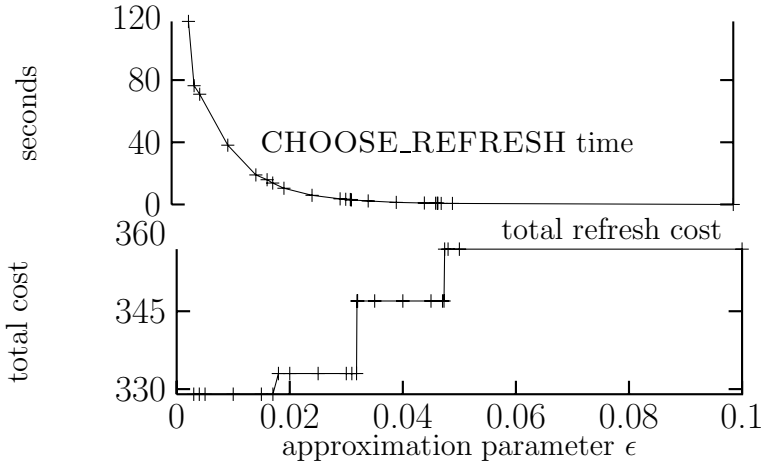


Figure 4.2: CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ time and refresh cost for varying ϵ.

were used as the bounds $[L_i, H_i]$, the closing value was used as the exact value $V(O_i)$, and the refresh cost C_i for each data object was set to a random number between 1 and 10. Running times were measured on a Sun Ultra-1 Model 140 running SunOS 5.6. In Figure 4.2 we fix the precision constraint $\delta = 100$ and vary ϵ in the knapsack approximation in order to plot CHOOSE_REFRESH time and total refresh cost of the selected tuples. Smaller values for ϵ increase the CHOOSE_REFRESH time but decrease the refresh cost. However, since the CHOOSE_REFRESH time increases quadratically while the refresh cost only decreases by a small fraction, it is not in general advantageous to set ϵ below 0.1 (which comes very close to optimal) unless refreshing is extremely expensive.

In Figure 4.3 we fix the approximation parameter $\epsilon = 0.1$ and vary δ in order to plot replica precision (precision constraint δ) versus communication performance (total refresh cost) for our CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ algorithm. This graph, a concrete instantiation of Figure 1.2 (b), clearly shows the continuous, monotonically decreasing tradeoff between precision and performance that characterizes approximate replication systems.

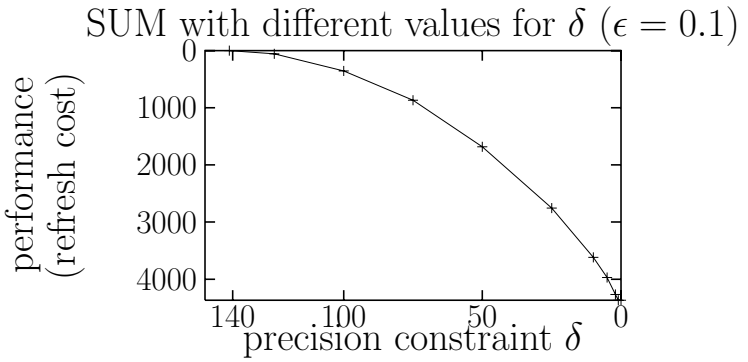


Figure 4.3: Precision-performance tradeoff for CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$.

4.3.4 Computing COUNT with No Selection Predicate

When no selection predicate is present, computing COUNT amounts to computing the cardinality of the table. Since we currently require all insertions and deletions to be propagated immediately to the data repository, the cardinality of the replicated copy of a table is always equal to the cardinality of the master copy, so there is no need for refreshes.

4.3.5 Computing AVG with No Selection Predicate

When no selection predicate is present, the procedure for computing the AVG aggregate is as follows. First, compute $COUNT$, which as discussed in Section 4.3.4 is simply the cardinality of the replicated T. Then, compute the bounded SUM as described in Section 4.3.3 with $\delta = \delta \cdot COUNT$ to produce $[L_{SUM}, H_{SUM}]$. Finally, let:

$$[L_A, H_A] = [\frac{L_{SUM}}{COUNT}, \frac{H_{SUM}}{COUNT}]$$

Since the bound width $H_A - L_A = \frac{H_{SUM} - L_{SUM}}{COUNT}$, by computing SUM such that $H_{SUM} - L_{SUM} \leq \delta \cdot COUNT$, we are guaranteeing that $H_A - L_A \leq \delta$, and the precision constraint is satisfied. The running time is dominated by the running time of the CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ algorithm, which is given in Section 4.3.3.

Consider query Q_3 from Section 4.1.2 to compute the average traffic level in the entire network, and let precision constraint $\delta = 10$. We first compute $COUNT = 6$,

and then compute SUM with $\delta = \delta \cdot COUNT = 10 \cdot 6 = 60$. The column labeled W' in Figure 4.1 shows the knapsack weight assigned to each tuple based on the replicated bounds for *traffic*. Using the optimal Knapsack algorithm, the SUM computation will cause tuples 5 and 6 to be refreshed, resulting in a bounded SUM of [618, 678]. Dividing by $COUNT = 6$ gives a bounded AVG of [103, 113].

4.4 Modifications to Incorporate Selection Predicates

When a selection predicate involving approximate replicas is present in the query, both computing bounded aggregate results and choosing refresh tuples to meet the precision constraint become more complicated. This section presents modifications to the algorithms in Section 4.3 to handle single-table aggregation queries with selection predicates. We begin by introducing techniques common to all aggregation queries with predicates, regardless of which aggregation function is present.

Consider a selection predicate involving at least one column of T that contains approximate replicas. The system can partition T into three disjoint sets: T^-, $T^?$, and T^+. T^- contains those tuples that cannot possibly satisfy the predicate given current bounded data. T^+ contains tuples that are guaranteed to satisfy the predicate given current bounded data. All other tuples are in $T^?$, meaning that there exist some exact values within the current bounds that will cause the predicate to be satisfied, and other values that will cause the predicate not to be satisfied. The process of classifying tuples into T^-, $T^?$, and T^+ when the selection predicate involves at least one column with approximate replicas is detailed next.

For examples in the remainder of this section we refer to Figure 4.4, which shows the classification for three different predicates over the data from Figure 4.1, both before and after the exact values are accessed by requesting refreshes.

	$(bandwidth > 50) \wedge (latency < 10)$		$latency > 10$		$traffic > 100$	
	before refresh	*after refresh*	*before*	*after refresh*	*before*	*after refresh*
1	T^+	T^+	T^-	T^-	$T^?$	T^-
2	$T^?$	T^+	T^-	T^-	T^+	T^+
3	T^-	T^-	T^+	T^+	$T^?$	T^+
4	$T^?$	T^+	$T^?$	T^-	T^+	T^+
5	$T^?$	T^-	$T^?$	T^+	$T^?$	T^-
6	$T^?$	T^-	T^-	T^-	$T^?$	T^+

Figure 4.4: Classification of tuples into T^-, $T^?$, and T^+ for three selection predicates.

4.4.1 Classifying Tuples by a Selection Predicate

Our algorithms for choosing tuples to refresh in the presence of a selection predicate require that we first classify all tuples in T as belonging to one of T^-, T^+, or $T^?$. Let P be the predicate in the user's query, which we assume is an arbitrary boolean expression involving binary comparisons. We define two transformations on predicate P. The *Possible* transformation yields an expression that finds tuples that could possibly satisfy the predicate based on bounded values. The *Certain* transformation yields an expression that finds tuples that are guaranteed to satisfy the predicate based on bounded values. We can apply $Certain(P)$ to find tuples in T^+, and $(Possible(P) \wedge \neg Certain(P))$ to find tuples in $T^?$. All other tuples are in T^-.

Since $Certain(P)$ and $Possible(P)$ are predicates to be evaluated on the tuples of table T, they must be expressed in terms of constants, attributes whose values are exact, and endpoints (denoted *min* and *max*) of attributes whose values are ranges. To handle expressions uniformly, we assume that all values are ranges: in the case of a constant value K (respectively an attribute A whose value is exact), we let $K_{min} = K_{max} = K$ (respectively $A_{min} = A_{max} = A$). Figure 4.5 gives a set of translation rules—primarily equivalences—specifying how boolean expressions are translated into *Certain* and *Possible*. These rules are applied recursively to the query's selection predicate P to obtain $Certain(P)$ and $Possible(P)$. Note that disjunction for *Certain* and conjunction for *Possible* are implications rather than equivalences. Thus, when we translate $Possible(E_1 \wedge E_2)$ into $Possible(E_1) \wedge Possible(E_2)$ we may classify a tuple into $T^?$ when it should really be in T^-. Also, when we translate $Certain(E_1 \vee E_2)$ into $Certain(E_1) \vee Certain(E_2)$ we may classify a tuple into $T^?$ when it should really

expression E	$Possible(E)$	$Certain(E)$
$[x_{min}, x_{max}] = [y_{min}, y_{max}]$	$\Leftrightarrow (x_{min} \leq y_{max})$ $\wedge (x_{max} \geq y_{min})$	$\Leftrightarrow x_{min} = x_{max}$ $= y_{min} = y_{max}$
$[x_{min}, x_{max}] < [y_{min}, y_{max}]$	$\Leftrightarrow x_{min} < y_{max}$	$\Leftrightarrow x_{max} < y_{min}$
$[x_{min}, x_{max}] \leq [y_{min}, y_{max}]$	$\Leftrightarrow x_{min} \leq y_{max}$	$\Leftrightarrow x_{max} \leq y_{min}$
$\neg E_1$	$\Leftrightarrow \neg Certain(E_1)$	$\Leftrightarrow \neg Possible(E_1)$
$E_1 \vee E_2$	$\Leftrightarrow Possible(E_1)$ $\vee Possible(E_2)$	$\Leftarrow Certain(E_1)$ $\vee Certain(E_2)$
$E_1 \wedge E_2$	$\Rightarrow Possible(E_1)$ $\wedge Possible(E_2)$	$\Leftrightarrow Certain(E_1)$ $\wedge Certain(E_2)$

Figure 4.5: Translation of range comparison expressions.

be in T^+. Cases where we misclassify tuples are extremely unusual (because they involve very special cases of correlation between subexpressions), and note that these misclassifications affect only the optimality and not the correctness of our algorithms.

We now illustrate how to use the rules in Figure 4.5 to derive expressions for $Certain(P)$ and $Possible(P)$ in terms of range endpoints. For the predicate $P = (bandwidth > 50) \wedge (latency < 10)$, $Certain(P)$ becomes $(bandwidth_{min} > 50) \wedge (latency_{max} < 10)$, and $Possible(P)$ becomes $(bandwidth_{max} > 50) \wedge (latency_{min} < 10)$. The column labeled "$(bandwidth > 50) \wedge (latency < 10)$ before refresh" of Figure 4.4 shows the resulting classification of tuples in our example data of Figure 4.1 into T^-, $T^?$, and T^+.

It turns out that this technique is part of a more general mathematical framework introduced in [71] for evaluating predicates over data objects that have a set of possible values (in our case, an infinite set of points along the range $[L_i, H_i]$). The following relationships translate the notation used in this dissertation into the notation from [71]: $T^+ = \|T\|_*$, $T^? = \|T\|^* - \|T\|_*$, $T^- = \overline{\|T\|^*}$.

In general, the selection predicate does not influence the evaluation of the aggregate. As we will see in Sections 4.4.2–4.4.6, the only information needed from the selection predicate is the classification of tuples into T^+, T^-, and $T^?$. However, a slight refinement can be made if the selection predicate is over the same column as the aggregation.[3] In this special case, each tuple t_i in $T^?$ has a restriction on actual

[3]More generally, the refinement applies if the selection predicate always restricts the value of the

value $V(O_i)$ imposed by the selection predicate, in addition to the bound $[L_i, H_i]$. For example, bound $[L_i, H_i] = [3, 8]$ has an additional restriction on $V(O_i)$ under the predicate < 5, if $V(O_i)$ is to contribute to the result. To take advantage of this additional restriction, the bounds $[L_i, H_i]$ for tuples in $T^?$ can be shrunk before they are input to the result computation or CHOOSE_REFRESH algorithm. For example, if we are aggregating *latency* under the predicate *latency* > 10, we can modify any lower bounds below 10 to 10 by using $[\max(L_i, 10), H_i]$ instead of $[L_i, H_i]$.

Filters over T that find the tuples in T^+ and $T^?$ can always be expressed as simple predicates over approximate replica interval endpoints, and all of our algorithms for computing bounded answers and choosing tuples to refresh examine only tuples in T^+ and $T^?$. Therefore, the classification can be expressed as SQL queries and optimized by the system, possibly incorporating specialized indexes as discussed in Chapter 7 (Section 7.1.2).

4.4.2 Computing MIN with a Selection Predicate

When a selection predicate is present, the bounded MIN answer is:

$$[L_A, H_A] = [\min_{t_i \in T^+ \cup T^?}(L_i), \min_{t_i \in T^+}(H_i)]$$

In the "worst case" for L_A, all tuples in $T^?$ satisfy the predicate (*i.e.*, they turn out to be in T^+), so the smallest lower bound of any tuple that might satisfy the predicate forms the lower bound for the answer. In the "worst case" for H_A, tuples in $T^?$ do not satisfy the predicate (*i.e.*, they turn out to be in T^-), so the smallest upper bound of the tuples guaranteed to satisfy the predicate forms the only guaranteed upper bound for the answer. In our running example, consider query Q_4: find the minimum *traffic* where (*bandwidth* > 50) $\wedge$ (*latency* < 10). The result using the data from Figure 4.1 and classifications from Figure 4.4 is [90, 105].

CHOOSE_REFRESH$_{\text{MIN}}$ chooses T_R to be exactly the tuples $t_i \in T^+ \cup T^?$ such that $L_i < \min_{t_k \in T^+}(H_k) - \delta$. This algorithm is essentially the same as

aggregation column. For example, the predicate $T.a < 5 \wedge T.b \neq 2$ always restricts the value of column $T.a$ to be less than 5.

CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$, and is correct and optimal for the same reason (see Section 4.3.1). The only additional case to consider is that refreshing tuples in $T^?$ may move them into T^-. However, such tuples do not contribute to the actual MIN, and thus do not affect the bound of the answer $[L_A, H_A]$. Hence, the precision constraint is still guaranteed to hold. As with CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$, the running time for CHOOSE_REFRESH$_{\text{MIN}}$ can be sublinear if B-tree indexes are available on both the upper and lower bounds. Otherwise, the worst-case running time for CHOOSE_REFRESH$_{\text{MIN}}$ is $O(n)$.

For our query Q_4 with precision constraint $\delta = 10$, CHOOSE_REFRESH$_{\text{MIN}}$ chooses $T_R = \{5, 6\}$, since tuples 5 and 6 may pass the selection predicate and their low values are less than $\min_{t_k \in T^+}(H_k) - \delta = 105 - 10 = 95$. After refreshing, tuples 5 and 6 turn out not to pass the selection predicate, so the bounded MIN is [95, 105].

4.4.3 Computing MAX with a Selection Predicate

The MAX aggregation function is symmetric to MIN. Thus:

$$[L_A, H_A] = [\max_{t_i \in T^+}(L_i), \max_{t_i \in T^+ \cup T^?}(H_i)]$$

and the CHOOSE_REFRESH$_{\text{MAX}}$ algorithm chooses T_R to be all tuples $t_i \in T^+ \cup T^?$ such that $H_i > \max_{t_k \in T^+}(L_k) + \delta$.

4.4.4 Computing SUM with a Selection Predicate

To compute SUM in the presence of a selection predicate:

$$[L_A, H_A] = [\sum_{t_i \in T^+} L_i + \sum_{t_i \in T^? \wedge L_i < 0} L_i, \sum_{t_i \in T^+} H_i + \sum_{t_i \in T^? \wedge H_i > 0} H_i]$$

The "worst case" for L_A occurs when all and only those tuples in $T^?$ with negative values for L_i satisfy the selection predicate and thus contribute to the result. Similarly, the "worst case" for H_A occurs when only tuples in $T^?$ with positive values for H_i satisfy the predicate.

The CHOOSE_REFRESH$_{\text{SUM}}$ algorithm is similar to CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$, which maps the problem to the 0/1 Knapsack Problem (Section 4.3.3). The following two modifications are required. First, we ignore all tuples $t_i \in T^-$. Second, for tuples $t_i \in T^?$, we set W_i to one of three possible values. If $L_i \geq 0$, let $W_i = H_i - 0 = H_i$. If $H_i \leq 0$, let $W_i = 0 - L_i = -L_i$. Otherwise, let $W_i = (H_i - L_i)$ as before. The idea is that we want to effectively extend the bounds for all tuples in $T^?$ to include 0, since it is possible that these tuples are actually in T^- and thus do not contribute to the SUM (*i.e.*, contribute value 0). In the knapsack formulation, to extend the bounds to 0 we need to adjust the weights as specified above.

4.4.5 Computing COUNT with a Selection Predicate

The bounded answer to the COUNT aggregation function in the presence of a selection predicate is:

$$[L_A, H_A] = [|T^+|, |T^+| + |T^?|]$$

For example, consider query Q_5 from Section 4.1.2 that asks for the number of links that have *latency* > 10. Figure 4.4 shows the classification of tuples into T^-, $T^?$, and T^+. Since $|T^+| = 1$ and $|T^?| = 2$, the bounded COUNT is [1, 3].

The CHOOSE_REFRESH$_{\text{COUNT}}$ algorithm is based on the fact that $H_A - L_A = |T^?|$, and that refreshing a tuple in $T^?$ is guaranteed to remove it from $T^?$. Given these two facts, the optimal CHOOSE_REFRESH$_{\text{COUNT}}$ algorithm is to let T_R be the $\lceil |T^?| - \delta \rceil$ cheapest tuples in $T^?$. Using a B-tree index on cost, this algorithm runs in sublinear time. Otherwise, the worst-case running time for CHOOSE_REFRESH$_{\text{COUNT}}$ requires a sort and is $O(n \cdot \log n)$.

Consider again query Q_5 and suppose $\delta = 1$. Since $|T^?| = 2$, CHOOSE_REFRESH$_{\text{COUNT}}$ selects $T_R = \{5\}$, which is the $\lceil |T^?| - \delta \rceil = \lceil 2 - 1 \rceil = 1$ cheapest tuple in $T^?$. After updating this tuple (which turns out to be in T^+), the bounded COUNT is [2, 3].

4.4.6 Computing AVG with a Selection Predicate

Computing the Bounded Answer

Computing the bounded AVG when a predicate is present is somewhat more complicated than computing the other aggregates. With a predicate, COUNT is an approximate replica as well as SUM, so it is no longer a simple matter of dividing the endpoints of the SUM bound by the exact COUNT value (as in Section 4.3.5). To compute the lower bound on AVG, we start by computing the average of the low endpoints of the T^+ bounds, and then average in the low endpoints of the $T^?$ bounds one at a time in increasing order until the point at which the average increases. Computing the upper bound on AVG is the reverse. For example, consider query Q_6 from Section 4.1.2 that asks for the average latency for links having *traffic* > 100. To compute the lower bound, we start by averaging the low endpoints of T^+ tuples 2 and 4, and then average in the low endpoints of $T^?$ tuples 1 and then 6 to obtain a lower bound on average latency of 5. We stop at this point since averaging in further $T^?$ tuples would increase the lower bound.

Formally, this computation is as follows. First, let $S_L = \sum_{t_i \in T^+} L_i$ and $K_L = |T^+|$, the sum and cardinality of the low values in T^+. Then, let A represent the tuples $t_i \in T^?$, sorted in increasing order by L_i. Let a be the first element of A. If $L_a < \frac{S_L}{K_L}$, then add L_a to S_L and 1 to K_L. Advance a and continue this process until $L_a \geq \frac{S_L}{K_L}$. Similarly, let $S_H = \sum_{t_i \in T^+} H_i$ and $K_H = |T^+|$. Now, let A represent the tuples $t_i \in T^?$, sorted in decreasing order by H_i. Let a be the first element of A. If $H_a > \frac{S_H}{K_H}$, then add H_a to S_H and 1 to K_H. Advance a and continue this process until $H_a \leq \frac{S_H}{K_H}$. Finally, let:

$$[L_A, H_A] = [\frac{S_L}{K_L}, \frac{S_H}{K_H}]$$

For example, consider query Q_6 from Section 4.1.2 that asks for the average *latency* for links having *traffic* > 100. First, we classify tuples into T^-, $T^?$, and T^+ as shown in Figure 4.4. Since $T^+ = \{2, 4\}$, initially $S_L = 14$ and $K_L = 2$. A is [1, 6, 5, 3], which are the tuples in $T^?$ sorted in increasing order by L_i. First, we let $a = 1$, and since $L_a = 2 < \frac{S_L}{K_L} = \frac{14}{2} = 7$, we set $S_L = S_L + L_a = 14 + 2 = 16$ and $K_L = K_L + 1 = 2 + 1 = 3$. Then, we let $a = 6$, and since $L_a = 4 < \frac{S_L}{K_L} = \frac{16}{3} = 5.3$,

we set $S_L = S_L + L_a = 16 + 4 = 20$ and $K_L = K_L + 1 = 3 + 1 = 4$. Next, we let $a = 5$, and note that $L_a = 8 \geq \frac{S_L}{K_L} = \frac{20}{4} = 5$, so we stop with $S_L = 20$ and $K_L = 4$. The computation of S_H and K_H proceeds similarly to yield $S_H = 34$ and $K_H = 3$. These results give a bounded AVG of $[\frac{S_L}{K_L}, \frac{S_H}{K_H}] = [\frac{20}{4}, \frac{34}{3}] = [5, 11.3]$. This algorithm for computing a tight bound for AVG has a running time of $O(n \cdot \log n)$.

A looser bound for AVG can be computed in linear time by first computing SUM as $[L_{SUM}, H_{SUM}]$ and COUNT as $[L_{COUNT}, H_{COUNT}]$ using the algorithms from Sections 4.4.4 and 4.4.5, then setting:

$$[L_A, H_A] = [\min(\frac{L_{SUM}}{H_{COUNT}}, \frac{L_{SUM}}{L_{COUNT}}), \max(\frac{H_{SUM}}{L_{COUNT}}, \frac{H_{SUM}}{H_{COUNT}})]$$

In our example, $[L_{SUM}, H_{SUM}] = [14, 55]$ and $[L_{COUNT}, H_{COUNT}] = [2, 6]$. Thus, the linear algorithm yields [2.3, 27.5]. Notice that this bound is indeed looser than the [5, 11.3] bound achieved by the $O(n \cdot \log n)$ algorithm above.

Choosing Tuples to Refresh

CHOOSE_REFRESH$_{\text{AVG}}$ is our most complicated scenario. We first give a very brief description and example, and then provide the full details.

Our CHOOSE_REFRESH$_{\text{AVG}}$ algorithm uses the fact that a loose bound on AVG can be achieved as a function of the bounds for SUM and COUNT, as in the linear algorithm described above. We choose refresh tuples that provide bounds for SUM and COUNT such that the bound for AVG as a function of the bounds for SUM and COUNT meets the precision constraint. This interaction is accomplished by using a modified version of the CHOOSE_REFRESH$_{\text{SUM}}$ algorithm that understands how the choice of refresh tuples for SUM affects the bound for COUNT. This algorithm sets a precision constraint for SUM that takes into account the changing bound for COUNT to guarantee that the overall precision constraint on AVG is met. CHOOSE_REFRESH$_{\text{AVG}}$ preserves the Knapsack Problem structure. Therefore, choosing refresh tuples for AVG can be accomplished by solving the 0/1 Knapsack Problem, and it has the same complexity as CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ (see Section 4.3.3).

In our example query Q_6 above, if we set $\delta = 2$ then CHOOSE_REFRESH$_{\text{AVG}}$ chooses a knapsack capacity of $M = 4$ and assigns a weight to each tuple as shown in the column labeled W'' in Figure 4.1. The knapsack optimally "contains" tuples 2 and 4. After refreshing the other tuples $T_R = \{1, 3, 5, 6\}$, the bounded AVG is [8, 9].

We now provide the full details of CHOOSE_REFRESH$_{\text{AVG}}$, which will guarantee that the precision constraint $H_A - L_A \leq \delta$ is satisfied with:

$$[L_A, H_A] = [\min(\frac{L_{SUM}}{H_{COUNT}}, \frac{L_{SUM}}{L_{COUNT}}), \max(\frac{H_{SUM}}{L_{COUNT}}, \frac{H_{SUM}}{H_{COUNT}})]$$

Although it would be desirable to find a CHOOSE_REFRESH algorithm that guarantees the precision constraint is satisfied for the exact bound $[L_A, H_A] = [\frac{S_L}{K_L}, \frac{S_H}{K_H}]$ described above, we have not yet succeeded in finding such an algorithm.

CHOOSE_REFRESH$_{\text{AVG}}$ chooses a set of tuples T_R such that after refreshing the tuples in T_R and computing $[L_{SUM}, H_{SUM}]$ and $[L_{COUNT}, H_{COUNT}]$, $\Delta AVG = H_{AVG} - L_{AVG} = \max(\frac{H_{SUM}}{L_{COUNT}}, \frac{H_{SUM}}{H_{COUNT}}) - \min(\frac{L_{SUM}}{H_{COUNT}}, \frac{L_{SUM}}{L_{COUNT}}) \leq \delta$. To make it possible to choose bounds for SUM and COUNT that will guarantee $\Delta AVG \leq \delta$, we must formulate ΔAVG as a function of $\Delta SUM = H_{SUM} - L_{SUM}$ and $\Delta COUNT = H_{COUNT} - L_{COUNT}$. Based on this function, CHOOSE_REFRESH$_{\text{AVG}}$ chooses an "approximately optimal" set of tuples T_R to refresh that gives values for ΔSUM and $\Delta COUNT$ such that the precision constraint $\Delta AVG \leq \delta$ is guaranteed to be met.

The relationship between $[L_{SUM}, H_{SUM}]$, $[L_{COUNT}, H_{COUNT}]$, and ΔAVG is:

$$\Delta AVG \leq RHS = \frac{\Delta SUM + (\frac{\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM})}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

To show this inequality, we consider three cases. In case 1, if $L_{SUM} \geq 0$, $\Delta AVG \leq \frac{H_{SUM}}{L_{COUNT}} - \frac{L_{SUM}}{H_{COUNT}}$, which gives:

$$\Delta AVG \leq RHS_1 = \frac{\Delta SUM + (\frac{H_{SUM}}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

In case 2, if $H_{SUM} \leq 0$, $\Delta AVG \leq \frac{H_{SUM}}{H_{COUNT}} - \frac{L_{SUM}}{L_{COUNT}}$, which gives:

$$\Delta AVG \leq RHS_2 = \frac{\Delta SUM + (\frac{-L_{SUM}}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

Otherwise, in case 3 ($L_{SUM} < 0$ and $H_{SUM} > 0$), $\Delta AVG \leq \frac{H_{SUM}}{L_{COUNT}} - \frac{L_{SUM}}{L_{COUNT}}$, which gives:

$$\Delta AVG \leq RHS_3 = \frac{\Delta SUM + (\frac{H_{SUM} - L_{SUM}}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

All three cases are equivalent to $RHS = \frac{\Delta SUM + (\frac{\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM})}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$. In case 1, $RHS_1 = RHS$ since $L_{SUM} \geq 0$ implies $\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM}) = H_{SUM}$. Similarly, in case 2, $RHS_2 = RHS$ since $H_{SUM} \leq 0$ implies $\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM}) = -L_{SUM}$. In case 3, $RHS_3 = RHS$ since the SUM bound straddles 0, which implies $\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM}) = H_{SUM} - L_{SUM}$.

Since our goal is to express ΔAVG as a function of ΔSUM and $\Delta COUNT$, we must eliminate all other values from the relationship:

$$\Delta AVG \leq \frac{\Delta SUM + (\frac{\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM})}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

To do this elimination, we substitute conservative estimates for the values L_{SUM}, H_{SUM}, and L_{COUNT}. Conservative estimates for these values are obtained by computing SUM and COUNT over the current replicated bounds as $[L'_{SUM}, H'_{SUM}]$ and $[L'_{COUNT}, H'_{COUNT}]$. Since, when the refreshes are performed, these bounds can shrink but not grow, $L'_{SUM} \leq L_{SUM}$, $H'_{SUM} \geq H_{SUM}$, and $L'_{COUNT} \leq L_{COUNT}$. Therefore, by examining the inequality relating $[L_{SUM}, H_{SUM}]$, $[L_{COUNT}, H_{COUNT}]$, and ΔAVG, it can be seen that substituting L'_{SUM} for L_{SUM}, H'_{SUM} for H_{SUM}, and L'_{COUNT} for L_{COUNT} makes the right-hand side strictly larger, so it is still an upper bound on ΔAVG. This substitution results in:

$$\Delta AVG \leq \mathcal{F}(\Delta SUM, \Delta COUNT) =$$

$$\frac{\Delta SUM + (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L'_{COUNT}}$$

Now that we finally have $\mathcal{F}(\Delta SUM, \Delta COUNT)$, an upper bound for ΔAVG as a function of ΔSUM and $\Delta COUNT$ (since L'_{SUM}, H'_{SUM}, and L'_{COUNT} are computed once and used as constants), we can substitute this function for ΔAVG in the precision constraint. Recall that the precision constraint requires that $\Delta AVG \leq \delta$. Substituting $\mathcal{F}(\Delta SUM, \Delta COUNT)$ for ΔAVG gives $\mathcal{F}(\Delta SUM, \Delta COUNT) \leq \delta$.

At this point, we have formulated the precision constraint in terms of only ΔSUM and $\Delta COUNT$. Rewriting the precision constraint in terms of ΔSUM gives:

$$\Delta SUM \leq L'_{COUNT} \cdot \delta - (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - \delta) \cdot \Delta COUNT$$

This formulation of the precision constraint can be used in place of the original constraint $\Delta AVG \leq \delta$. Therefore, the CHOOSE_REFRESH$_{\text{AVG}}$ algorithm is free to choose any values for ΔSUM and $\Delta COUNT$ that satisfy the reformulated precision constraint. We have thus reduced the task of choosing refresh tuples for AVG to the task of choosing refresh tuples for SUM under this reformulated constraint.

Normally, to choose refresh tuples for SUM, we have the constraint $\Delta SUM \leq \delta_{SUM}$. In this case, we instead have the constraint $\Delta SUM \leq L'_{COUNT} \cdot \delta - (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - \delta) \cdot \Delta COUNT$, so we let δ_{SUM} be the following function of $\Delta COUNT$:

$$\delta_{SUM}(\Delta COUNT) = L'_{COUNT} \cdot \delta - (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - \delta) \cdot \Delta COUNT$$

To see how to choose refresh tuples for SUM when δ_{SUM} is a function of $\Delta COUNT$, first recall that the CHOOSE_REFRESH$_{\text{SUM}}$ algorithm chooses refresh tuples for SUM by mapping it to the 0/1 Knapsack Problem, where the knapsack capacity $M = \delta_{SUM}$. Therefore, for the CHOOSE_REFRESH$_{\text{AVG}}$ algorithm, we need to make the knapsack capacity a function of $\Delta COUNT$. On the surface, it looks as

though this modification is not possible since there is no way to make the knapsack capacity a function instead of a constant. Fortunately, making the knapsack capacity a function of $\Delta COUNT$ is possible to fake. First, recall that $\Delta COUNT$ is equal to the number of tuples in $T^?$ that do not get refreshed to obtain their exact values (and thus remain in $T^?$ after the refreshes are performed). Also, recall that the set of items placed in the knapsack corresponds to $\overline{T_R}$: the set of tuples that will not be refreshed. It follows that $\Delta COUNT$ is equal to the number of $T^?$ tuples in the knapsack. Therefore, when the knapsack is empty, $\Delta COUNT = 0$ and thus the initial knapsack capacity $M = \delta_{SUM}(0) = L'_{COUNT} \cdot \delta$. Furthermore, every time a $T^?$ tuple is added to the knapsack, $\Delta COUNT$ increases by 1. Since the function $\delta_{SUM}(\Delta COUNT)$ is a line with (negative) slope:

$$m = -(\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - \delta)$$

the capacity of the knapsack decreases by the amount $-m$ every time a $T^?$ tuple is added to the knapsack. Observe that decreasing the knapsack capacity when an item is added is equivalent to increasing the weight of the item. Therefore, to simulate shrinking the knapsack by $-m$ every time $\Delta COUNT$ increases by 1, all we have to do is add the quantity $-m$ to the weight of each tuple in $T^?$.

To summarize, the CHOOSE_REFRESH$_{\text{AVG}}$ algorithm is exactly the same as the CHOOSE_REFRESH$_{\text{SUM}}$ algorithm (which maps to the 0/1 Knapsack Problem) with the following modifications: $M = L'_{COUNT} \cdot \delta$, and for all tuples $t_i \in T^?$, $W_i = W_i + (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - \delta)$. The values L'_{SUM}, H'_{SUM}, and L'_{COUNT} are found by computing SUM and COUNT over the current replicated bounds as $[L'_{SUM}, H'_{SUM}]$ and $[L'_{COUNT}, H'_{COUNT}]$. The running time of CHOOSE_REFRESH$_{\text{AVG}}$ is dominated by the running time of CHOOSE_REFRESH$_{\text{SUM}}$, which is given in Section 4.4.4.

Consider query Q_6 that asks for the average *latency* for links having *traffic* > 100, with $\delta = 2$. First, we classify tuples into T^-, $T^?$, and T^+, as shown in Figure 4.4. Then, we compute $[L'_{SUM}, H'_{SUM}] = [14, 55]$ and $[L'_{COUNT}, H'_{COUNT}] = [2, 6]$. We use these values to assign a weight to each tuple by computing the weight

used in the CHOOSE_REFRESH$_{\text{SUM}}$ algorithm, and for tuples in $T^?$, adding $\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - \delta = \frac{55}{2} - 2 = 25.5$. The column labeled W'' in Figure 4.1 shows these weights. Using the Knapsack Problem with $M = L'_{COUNT} \cdot \delta = 2 \cdot 2 = 4$, the knapsack optimally "contains" tuples 2 and 4. After refreshing the other tuples $T_R = \{1, 3, 5, 6\}$, the bounded AVG is [8, 9].

4.5 Aggregation Queries with Joins

Computing the bounded answer to an aggregation query with a join expression (*i.e.*, with multiple tables in the `FROM` clause) is no different from doing so with a selection predicate: in most SQL queries, a join is expressed using a selection predicate that compares columns of more than one table. Our method for determining membership of tuples in T^+, $T^?$, and T^- applies to join predicates as well as selection predicates. As before, the classification can be expressed as SQL queries and optimized by the system to use standard join techniques, possibly incorporating specialized indexes as discussed in Section 7.1.2.

On the other hand, choosing tuples to refresh is significantly more difficult in the presence of joins. First, since there are several "base" tuples contributing to each "aggregation" (joined) tuple, we can choose to refresh any subset of the base tuples. Each subset might shrink the answer bound by a different amount, depending how it affects the T^+, $T^?$, T^- classification combined with its effect on the aggregation column. Second, since each base tuple can potentially contribute to multiple aggregation tuples, refreshing a base tuple for one aggregation tuple can also affect other aggregation tuples. These interactions make the problem quite complex. We have considered various heuristic algorithms that choose tuples to refresh for join queries. As future work we plan to study the exact complexity of the problem, and we hope to find an approximation algorithm with a tunable ϵ parameter, as in the approximation algorithm for CHOOSE_REFRESH$_{\text{SUM}}$.

4.6 Related Work

We provided an overview of related work in Section 1.5 and discussed work specifically related to approximate replicas in Section 3.7. This chapter focused on one-time aggregation queries with selection predicates, which brings up a small amount of additional related work. Both [58] and [96] consider aggregation queries with selections. The APPROXIMATE approach [58] produces bounded answers when time does not permit the selection predicate to be evaluated on all tuples. However, APPROXIMATE does not deal with queries over bounded data. The work in [96] deals with queries over fuzzy sets. While approximate replicas can be considered as infinite fuzzy sets, this representation is not practical. Furthermore, the approach in [96] does not consider fuzzy sets as approximations of exact values available for a cost.

4.7 Chapter Summary

In this chapter we studied the problem of minimizing communication cost (*i.e.*, maximizing performance) in the presence of unanticipated one-time aggregation queries with precision constraints and optional selection predicates. We devised efficient algorithms for minimizing accesses to remote master copies that handle the five standard relational aggregation functions with optional selection predicates.

Chapter 5

Managing Precision in the Presence of One-Time Queries

5.1 Introduction

In the previous chapter we studied the problem of minimizing the communication cost incurred to answer ad-hoc, one-time queries with precision constraints by accessing a minimum-cost subset of source copies. Clearly, a significant factor determining the cost to evaluate one-time queries with precision constraints is the precision of data replicas maintained in the central repository. If more precise replicas are maintained at the repository, then one-time queries tend to access fewer exact source values, incurring lower communication cost to answer queries. However, as we have seen in Chapter 3, maintaining more precise replicas requires more frequent refreshes to keep up with changes in the exact source value, leading to higher communication cost for regular refreshes. It is not obvious how best to balance these two opposing effects.

When one-time queries are infrequent, and the workload is dominated by continuous queries, the precision of replicas accessed by one or more continuous queries can be set using the techniques of Chapter 3, and replication of other data objects is unnecessary since they can be fetched on demand when one-time queries are issued. However, in situations where one-time queries are frequent, it may be appropriate to replicate data objects not involved in any continuous queries to lower the cost to

answer one-time queries over them. In this chapter we study the problem of deciding what precision levels to use for approximate replicas of objects not involved in continuous queries but subject to intermittent accesses by one-time queries.

5.1.1 Chapter Outline

The remainder of the chapter is structured as follows. First we describe the effect of replica precision on overall communication cost incurred to answer one-time queries in terms of both source accesses and regular refreshes, and motivate the need to set precision adaptively in the presence of one-time queries. Then in Section 5.2 we describe our adaptive precision-setting algorithm. We justify our algorithm mathematically in Section 5.3. In Section 5.4 we describe our simulation environment and test data sets, then present our performance results. We first justify empirically our claim from Section 5.3 that our algorithm converges to optimal performance, by considering steady-state synthetic data. We then switch to real-world data, finding the best parameter settings to maximize the performance of our algorithm under dynamically changing conditions. Lastly we demonstrate that our algorithm performs as well as previous algorithms in the special case of exact replication, and outperforms exact replication algorithms when exact precision is not required. Related work specific to this chapter is discussed in Section 5.5, and a chapter summary is provided in Section 5.6.

5.1.2 Precision of Approximate Replicas

Recall that the central data repository maintains a numeric bound $[L_i, H_i]$ on each approximately replicated data object O_i. Replica precision is quantified as the inverse of the width of the bound (*i.e.*, $\textit{precision} = \frac{1}{H_i - L_i}$). At one extreme, a zero-width bound contains only the exact value and thus has infinite precision. In the other extreme, a bound of infinite width gives no information about the exact value and thus has zero precision. We assume that bounds remain constant until a refresh occurs. Although bounds that vary as a function of time are more general, we have found empirically that they are not particularly helpful, as discussed in Section 5.4.5.

Whenever the precision of a replicated bound is not adequate for a query running at the data repository, *i.e.*, the bound is too wide, the query initiates a refresh by requesting the exact value from the source, as discussed in Chapter 4. The source responds with the current exact value as well as a new bound $[L_i', H_i']$ to be used by subsequent queries. The cost incurred during a *query-initiated refresh* will be denoted C_{qr}. A *value-initiated refresh* incurs cost C_{vr}, and occurs whenever the exact value $V(O_i)$ at a data source exceeds its bound $[L_i, H_i]$ in the central repository. (In previous chapters we dealt either uniquely with value-initiated refreshes (Chapter 3) or query-initiated refreshes (Chapter 4) and in each case assumed a single per-object refresh cost of C_i. In this chapter we consider the two types of refreshes together, while treating each object independently, so we frame our discussion and analysis in terms of C_{vr} and C_{qr} for a single object.) Notice that value-initiated refreshes are never required for bounds of infinite width ($H_i - L_i = \infty$). Conversely, when a bound has zero width ($H_i - L_i = 0$), then a value-initiated refresh occurs every time $V(O_i)$ changes.

5.1.3 Adjusting Bound Widths

Both types of refreshes (value- and query-initiated) provide an opportunity for the source to adjust the bound being replicated. For now let us assume that whenever the source provides a new bound to the repository, the bound is centered around the current exact value. (Uncentered bounds are considered in Section 5.4.5, and like time-varying bounds they usually turn out not to be helpful.) Therefore, an approximation for a value $V(O_i)$ is uniquely determined by the bound width $W_i = H_i - L_i$. The objective in selecting a good bound width is to avoid the need for future refreshes, since we want to minimize communication cost. To avoid value-initiated refreshes, the bound should be wide enough to make it unlikely that modifications to the exact value will exceed the bound. On the other hand, to avoid query-initiated refreshes, the bound should be as narrow as possible. Since decreasing the chance of one type of refresh increases the chance of the other, it is not obvious how best to choose a bound width that minimizes the total probability that a refresh will be

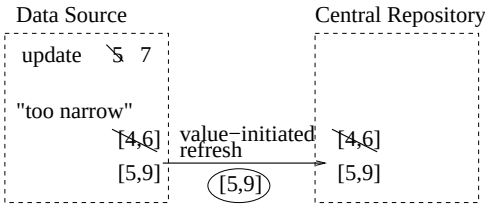
required.

Both of the factors that affect the choice of bound width—the variation of data values (which causes value-initiated refreshes) and the precision requirements of user queries (which cause query-initiated refreshes)—are difficult to predict, so we propose an adaptive algorithm that adjusts the width W as conditions change. We will present a parameterized algorithm for adjusting the precision of replicated approximations adaptively to achieve the best performance for one-time queries as data values, precision requirements, and/or overall one-query workload vary. The specific problem we address considers bound approximations to numeric values, but our ideas can be extended to other kinds of data and approximations, as discussed briefly in Chapter 7.

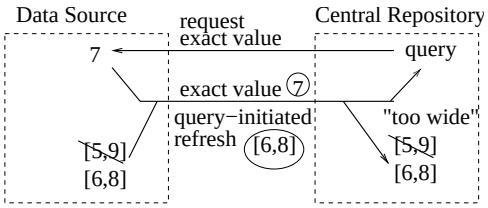
Our algorithm adjusts the precision of each approximate replica independently, and it strictly generalizes previous adaptive replication algorithms for exact copies (*e.g.*, [119]): we can set parameters to require that all approximations be exact, in which case our algorithm dynamically chooses whether or not to replicate each data value. We have implemented our algorithm and performed tests over synthetic and real-world data. We report a number of experimental results, which show the effectiveness of our algorithm at maximizing performance, and also show that in the special case of exact replication our algorithm performs as well as previous algorithms. In cases where it is acceptable for queries to produce answers with bounded imprecision, our algorithm easily outperforms previous algorithms for exact replication.

5.2 Precision-Setting Algorithm

Our overall strategy for setting bound widths adaptively in the presence of a workload of one-time queries is as follows. First start with some value for W. Each time a value-initiated refresh occurs (a signal that the bound was too narrow), increase W when sending the new bound. Conversely, each time a query-initiated refresh occurs (a signal that the bound was too wide), decrease W. This strategy, illustrated in Figure 5.1, finds a middle ground between very wide bounds that the value never exceeds yet are exceedingly imprecise, and very narrow bounds that are precise but need to be refreshed constantly as the value fluctuates.



(a) Growing the bound on a value-initiated refresh



(b) Shrinking the bound on a query-initiated refresh

Figure 5.1: Adaptive precision-setting algorithm for one-time queries.

We now define our algorithm precisely. The algorithm relies on five parameters as follows. The first two are functions of the particular distributed replication environment, while the remaining three can be set to tune the algorithm.

(1) the value-initiated refresh cost C_{vr}

(2) the query-initiated refresh cost C_{qr}

(3) the *adaptivity parameter* $\alpha \geq 0$

(4) the *lower threshold* $\tau_0 \geq 0$

(5) the *upper threshold* $\tau_\infty \geq 0$

These parameters and others we will introduce later are summarized in Table 5.1. Let us define a *cost factor* ρ as $\rho = 2 \cdot \frac{C_{vr}}{C_{qr}}$. The cost factor is based on the ratio of the two refresh costs and is used to determine how often to grow and shrink the bound width W as refreshes occur. The mathematical justification for multiplying the ratio by 2 is given in Section 5.3.

Our algorithm sets the new width W' for a refreshed bound based on the old width W during each value- or query-initiated refresh as follows. Recall that $\alpha \geq 0$ is the adaptivity parameter.

Symbol	*Meaning*	*Note*
C_{vr}	cost of a value-initiated refresh	used to determine cost factor ρ
C_{qr}	cost of a query-initiated refresh	used to determine cost factor ρ
ρ	cost factor defined as $2 \cdot \frac{C_{vr}}{C_{qr}}$	determines width adjustment probability
$\mathcal{C}$	overall cost (per time step)	metric our algorithm minimizes
W	bound width	set adaptively by our algorithm
W^*	width that minimizes $\mathcal{C}$	our algorithm converges to $W = W^*$
α	adaptivity parameter	how much to adjust width
τ_0	lower threshold	widths below τ_0 are set to 0
τ_∞	upper threshold	widths above τ_∞ are set to ∞
P_{vr}	probability of value-init. refresh	increases with precision
P_{qr}	probability of query-init. refresh	decreases with precision
δ	precision constraint of a query	parameter to experiments
δ_{avg}	avg. precision constraint of queries	parameter to experiments
$\Delta\delta$	variation of precision constraints	parameter to experiments
δ_{min}	minimum precision constraint	derived from δ_{avg} and $\Delta\delta$
δ_{max}	maximum precision constraint	derived from δ_{avg} and $\Delta\delta$
n	number of data sources	parameter to experiments
κ	repository size (in # of objects)	parameter to experiments
T_q	time period between queries	parameter to experiments
s	random walk step size	used for analysis

Table 5.1: One-time query precision-setting model and algorithm symbols.

- value-initiated refresh:
  with probability $\min\{\rho, 1\}$, set $W' \leftarrow W \cdot (1 + \alpha)$
- query-initiated refresh:
  with probability $\min\{\frac{1}{\rho}, 1\}$, set $W' \leftarrow \frac{W}{(1+\alpha)}$

In Section 5.3 we will justify mathematically why these are the optimal probability settings for width adjustment. Intuitively, the idea is to continually adapt the bound width to balance the likelihood of the two types of refreshes. However, if two value-initiated refreshes are more expensive than one query-initiated refresh, *i.e.*, $\rho > 1$, a larger width is preferred, so the width is not decreased on every query-initiated refresh. Conversely, if one query-initiated refresh is more expensive than two value-initiated refreshes, *i.e.*, $\rho < 1$, a smaller width is preferred, so the width is not increased on every value-initiated refresh. Whenever the width is adjusted, the magnitude of the

adjustment is controlled by the adaptivity parameter α.

Now let us consider the lower and upper thresholds, τ_0 and τ_∞. When our algorithm computes a bound width $W < \tau_0$, we instead set $W = 0$, and when we compute a $W \geq \tau_\infty$, we instead set $W = \infty$. The purpose of these thresholds is to accommodate boundary cases where either exact replication ($W = 0$) or no replication ($W = \infty$) is appropriate, since without this mechanism the width would never actually reach these extreme values. The source still retains the original width, and uses it when setting the next width W'. As part of our performance study we describe how to set these parameters and others.

5.3 Justification of Algorithm

In this section we justify our algorithm mathematically. Let P_{vr} and P_{qr} represent the probability that a value- or query-initiated refresh (respectively) will occur at each time step. Then the expected *overall cost* per time step $\mathcal{C} = C_{vr} \cdot P_{vr} + C_{qr} \cdot P_{qr}$, where C_{vr} and C_{qr} are the costs of value- and query-initiated refreshes (respectively) as introduced in Section 5.2.

For a given replicated approximation with width W, the probability of each type of refresh can be written as $P_{vr} = \frac{K_1}{W^2}$ and $P_{qr} = K_2 \cdot W$, where K_1 and K_2 are *model parameters* that depend on the nature of the data and updates, the frequency of queries, and the distribution of query precision requirements. Next, in Section 5.3.1, we justify these equations in detail for the case of bound approximations. However, the important—and intuitive—point is that, for all kinds of data and approximations, the value-initiated refresh probability increases with precision (*i.e.*, with a smaller W), while the query-initiated refresh probability decreases with precision.

5.3.1 Estimating the Probability of Refresh for One-Time Queries

To determine the probability of each type of refresh, let us consider a simplified model. First, we model the changing data value as a random walk in one dimension. In the

random walk model the value either increases or decreases by a constant amount s at each time step. A value-initiated refresh occurs when the value moves out of the replicated bound. Queries access the data every T_q time steps. We assume each query accesses only one data value and is accompanied by a *precision constraint* δ sampled from a uniform distribution between 0 and δ_{max}. A query-initiated refresh occurs when the replicated bound's width is larger than a query's precision constraint δ. Although this model is simplified, it is useful for deriving formulas and demonstrating the principles behind our algorithm. As we show empirically in Section 5.4, our algorithm works well for real-world data under a variety of query workloads and precision constraints.

Using our model, we now derive expressions for the value- and query-initiated refresh probabilities at each time step, P_{vr} and P_{qr} respectively. These probabilities depend on the nature of the data and updates, the frequency of queries, and the distribution of precision constraints. The probability of a query-initiated refresh at a given time step equals the probability P_q that a query is issued, multiplied by the probability $P_{\delta<W}$ that the precision of the replicated bound does not meet the precision constraint of the query. Clearly, the probability P_q that a query occurs at each time step is $P_q = \frac{1}{T_q}$. Recall that in our model, precision constraints are uniformly distributed between 0 and δ_{max}. Thus, as long as $0 \leq W \leq \delta_{max}$, $P_{\delta<W} = \frac{W}{\delta_{max}}$. Putting it all together, we have $P_{qr} = P_q \cdot P_{\delta<W} = \frac{W}{T_q \cdot \delta_{max}}$. Therefore, the probability of a query-initiated refresh is proportional to the bound width: $P_{qr} \propto W$.

Determining a formula for the value-initiated refresh probability requires an analysis of the behavior of a random walk. In the random walk model, after t steps of size s, the probability distribution of the value is a binomial distribution with variance $s^2 \cdot t$ [44]. Chebyshev's Inequality [44] gives an upper bound on the probability P that the value is beyond any distance k from the starting point: $P \leq t \cdot (\frac{s}{k})^2$. If we let $k = \frac{W}{2}$, and treat the upper bound as a rough approximation, we have the probability P_{vr} that the value has exceeded its bound after t time steps: $P_{vr} \approx t \cdot (\frac{2 \cdot s}{W})^2$. Therefore, the probability of a value-initiated refresh is proportional to the reciprocal of the square of the bound width: $P_{vr} \propto \frac{1}{W^2}$.

5.3.2 Minimizing Overall Refresh Cost

Now that we know how P_{vr} and P_{qr} depend on W, we can rewrite our cost per time step $\mathcal{C}$ in terms of W: $\mathcal{C}(W) = C_{vr} \cdot \frac{K_1}{W^2} + C_{qr} \cdot K_2 \cdot W$. Our goal then is to find the value for W that minimizes this expression, which is achieved by finding the root of the derivative. Using this approach, the optimal value for W is $W^* = (2 \cdot \frac{C_{vr}}{C_{qr}} \cdot \frac{K_1}{K_2})^{(\frac{1}{3})} = (\rho \cdot \frac{K_1}{K_2})^{(\frac{1}{3})}$, where $\rho = 2 \cdot \frac{C_{vr}}{C_{qr}}$ is the cost factor we defined in Section 5.2. Unfortunately, setting the bound width W based on this formula for W^* is difficult unless update behaviors and query/update workloads are stable and known in advance, since model parameters K_1 and K_2 depend on these factors. One approach is to monitor these factors at run-time to set K_1 and K_2 appropriately, which is similar to the approach taken by Divergence Caching [54]. However, we will see that our approach achieves the same optimal bound width W^* without the monitoring complexity or overhead.

Our approach is motivated by the observation that $\rho \cdot P_{vr} = P_{qr}$ when $W = W^* = (\rho \cdot \frac{K_1}{K_2})^{(\frac{1}{3})}$. Let us first consider the special case where $\rho = 1$. (We will discuss other values for ρ momentarily.) In this special case, the optimal value for W occurs exactly when the two types of refreshes are equally likely. Our algorithm takes advantage of this observation by dynamically adjusting the bound width W so as to equate the likelihood of each type of refresh, thereby discovering the optimal width W^*. Our algorithm adjusts the width W based solely on observing refreshes as they occur, without the need for storing history or for direct measurements of updates, queries, or precision requirements. Furthermore, as conditions change over time, our algorithm adapts to always move W toward the optimal width W^*. The *adaptivity parameter* $\alpha \geq 0$, introduced in Section 5.2, controls how quickly the algorithm is able to adapt to changing conditions.

The graph in Figure 5.2 illustrates the principle behind our algorithm. Still considering $\rho = 1$, it plots the overall cost per time $\mathcal{C}$ and refresh probabilities P_{vr} and P_{qr} as functions of the bound width W. The model parameters K_1 and K_2 are fixed as $K_1 = 1$ and $K_2 = \frac{1}{200}$. (These values were set based roughly on a query period of 10 seconds and an average precision constraint of 10. Changing these values only shifts the graph.) Notice that the width W^* that minimizes the overall cost $\mathcal{C}$ corresponds exactly to the point where the curves for P_{vr} and P_{qr} cross. Therefore, by equalizing

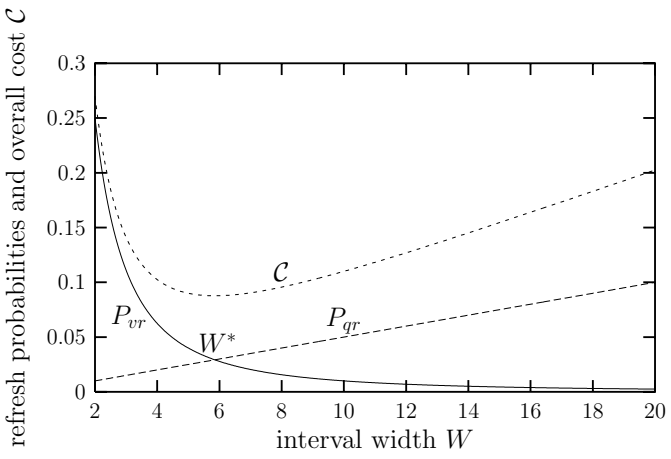


Figure 5.2: Cost and refresh probabilities when $\rho = 1$ as functions of bound width.

the chance that refreshes will be either value- or query-initiated, the optimal width W^* is discovered.

Now consider the general case where ρ can have any value. Our algorithm still discovers the optimal width W^*. Recall that the optimal width occurs when $\rho \cdot P_{vr} = P_{qr}$. Our algorithm achieves this condition by not always adjusting the bound width on every refresh. In cases where $\rho < 1$, it is desirable for value-initiated refreshes to be more likely than query-initiated refreshes. Thus, the width is decreased on every query-initiated refresh but only increased with probability ρ on value-initiated refreshes. Conversely, in cases where $\rho > 1$, the width is increased on every value-initiated refresh but only decreased with probability $\frac{1}{\rho}$ on query-initiated refreshes.

5.3.3 Verification of Convergence to Optimal Width

To verify that our algorithm converges on the optimal width for any value of ρ, we analyze the way our algorithm adjusts the bound width over time. First, observe that over the course of Δt time units, assuming constant per-unit-time query- and value-initiated refresh probabilities P_{vr} and P_{qr}, our algorithm will update the bound width from a starting value of W to a new value of W' as follows:

$$W' = W \cdot (1+\alpha)^{\Delta t \cdot (P_{vr} \cdot \min\{\rho,1\} - P_{qr} \cdot \min\{1/\rho,1\})}$$

Equivalently, we write:

$$\frac{\Delta W}{\Delta t} = W \cdot \frac{(J^{\Delta t} - 1)}{\Delta t}$$

where $\Delta W = W' - W$ and $J = (1+\alpha)^{P_{vr}\cdot\min\{\rho,1\} - P_{qr}\cdot\min\{1/\rho,1\}}$.

For the purpose of studying the convergence properties of our algorithm, let us for the moment consider refreshes and width adjustments as continuous processes rather than discrete events, as they really are. Taking the limit of the ratio $\frac{\Delta W}{\Delta t}$ as $\Delta t \to 0$, we find:

$$\frac{\partial W}{\partial t} = W \cdot \ln J$$

Substituting our formulae for the value- and query-initiated refresh probabilities as functions of the current bound width W, $P_{vr} = \frac{K_1}{W^2}$ and $P_{qr} = K_2 \cdot W$, we arrive at the following expression showing how our algorithm adjusts bound width over time in a continuous scenario:

$$\frac{\partial W}{\partial t} = \ln(1+\alpha) \cdot \left(\frac{K_1}{W} \cdot \min\{\rho, 1\} - K_2 \cdot W^2 \cdot \min\{1/\rho, 1\} \right)$$

According to this model, whenever the current width W is greater than the optimal width W^* (*i.e.*, $W > W^* = (\rho \cdot \frac{K_1}{K_2})^{(\frac{1}{3})}$), $\frac{\partial W}{\partial t} < 0$ for any value of ρ, so our algorithm tends to decrease the width toward the optimal one. Conversely, whenever $W < W^*$, $\frac{\partial W}{\partial t} < 0$ so our algorithm tends to increase the width toward the optimal one. Finally, when the optimal width W^* is achieved, $\frac{\partial W}{\partial t} = 0$ and our algorithm does not alter the width, as desired. Of course, since in our actual algorithm W cannot be updated continually and instead is adjusted in discrete jumps, the end result may only converge near the optimal point, which in any case we do not expect to be a fixed target. Our experimental results, reported next, confirm that our algorithm performs well on real data.

5.4 Performance Study

In this section we present the results of a performance study of our algorithms, and of related algorithms, using synthetic data as well as real-world data taken from a network monitoring application. We first describe our simulation environment in Section 5.4.1. In Section 5.4.2 we present results demonstrating empirically that our algorithm does achieve optimal performance in the steady state, as motivated mathematically in Section 5.3. We introduce our real-world data set in Section 5.4.3, and in Section 5.4.4 we present results indicating how best to set the tunable parameters of our algorithm. In Section 5.4.5 we discuss some variations of our algorithm that proved to be unsuccessful in most cases. Finally, in Section 5.4.6 we show that our algorithm precisely matches the performance of adaptive exact replication when exact precision is required, and outperforms exact replication when exact precision is not required.

5.4.1 Simulator Description

To study our adaptive algorithm empirically, we built a discrete event simulator of an environment with n data sources and one central repository. Each source holds one exact numeric value, and the repository can hold up to $\kappa \leq n$ bound approximations to exact source values. In our synthetic experiments, exact values are updated every time unit (which we set to be one second) with a specified update distribution. In our real-world experiments, the timing and values of updates are generated from the network performance data we are using. For both types of experiments, a query is executed at the repository every T_q seconds. We will describe how queries and query precision requirements are generated momentarily, but it is important to note that our algorithm is not specialized to any particular type of query. The only assumption made by our algorithm is that for each approximate value, the probability of a query-initiated refresh is proportional to the width of the approximation.

The queries we generate attempt to balance generality and practicality and correspond to the bounded aggregation queries of Chapter 4. Each query asks for either the SUM or MAX of a set of approximate values in the repository, where the query

result is itself a bound approximation. Each query is accompanied by a *precision constraint* $\delta \geq 0$ specifying the maximum acceptable width of the result. Precision constraints are generated based on parameters δ_{avg} (average precision constraint) and $\Delta\delta$ (precision constraint variation): they are sampled from a uniform distribution between $\delta_{min} = \delta_{avg} \cdot (1 - \Delta\delta)$ and $\delta_{max} = \delta_{avg} \cdot (1 + \Delta\delta)$. Using the algorithms presented in Chapter 4, from the query type, precision constraint, and approximate data, a (possibly empty) subset of the approximations are selected for query-initiated refresh, after which the desired precision for the query result is guaranteed. Again, it is important to note that our algorithm is not aware of or tuned to these query types or this method of expressing precision requirements—they are used solely to generate a realistic and interesting query load.

5.4.2 Optimality of Algorithm

Our first experiment was a simple one to verify the correctness of our basic model and the optimality of our algorithm on steady-state data. We used synthetic data consisting of only one source data item, whose value performs a random walk in one dimension: every second, the value either increases or decreases by an amount sampled uniformly from $[0.5, 1.5]$. We simulated a workload having query period $T_q = 2$ seconds, average precision constraint $\delta_{avg} = 20$, and precision constraint variation $\Delta\delta = 1$, in an environment with $\rho = 1$. The query type (SUM or MAX) is irrelevant since we have only one data item.

Our goal was to establish the correctness of our model for the refresh probabilities P_{vr} and P_{qr}, *i.e.*, to show that as the data undergoes a random walk P_{vr} and P_{qr} are proportional to $\frac{1}{W^2}$ and W respectively, and for $\rho = 1$, equalizing P_{vr} and P_{qr} maximizes performance. Thus, we fixed bound width W for each run (*i.e.*, we turned off the part of our algorithm that adjusts widths dynamically), but varied W across runs. We measured the average periods with which value- and query-initiated refreshes occurred, taking their reciprocals to obtain P_{vr} and P_{qr}, respectively. Measurements taken during an initial warm-up period were discarded, as in all subsequent reported experiments.

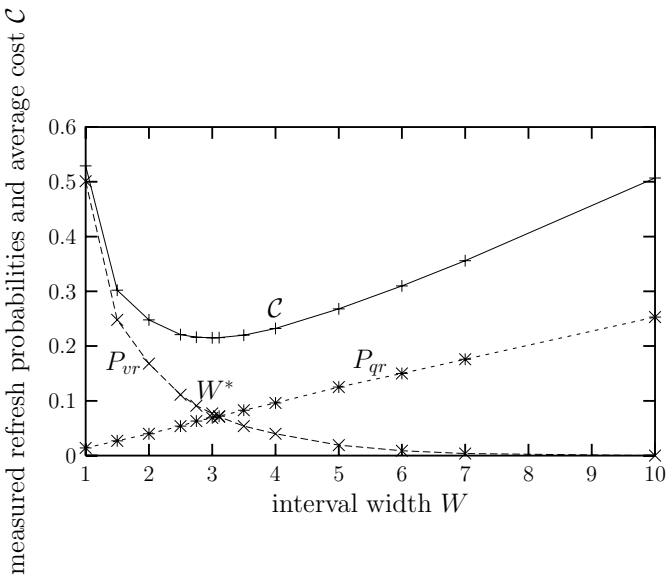


Figure 5.3: Measured cost and refresh probabilities when $\rho = 1$ as functions of bound width.

The results are shown in Figure 5.3, which bears a striking resemblance to Figure 5.2. The measured values for P_{vr} and P_{qr} are indeed proportional to $\frac{1}{W^2}$ and W, respectively, and the measured overall cost per time $\mathcal{C}$ for different values of W also is shown in Figure 5.3. From this graph we verify empirically that, in the $\rho = 1$ case, the minimum overall cost does indeed occur when the two refresh probabilities are equal.

We then ran the same experiment letting our algorithm adjust bound widths. The algorithm converged to $W = 3.11$, resulting in performance within 1% of the optimal W^* shown in Figure 5.3. We further evaluated the optimality of our algorithm with all combinations of $T_q \in \{1, 2\}$, $\delta_{avg} \in \{10, 20\}$, and $\rho \in \{1, 4\}$. In all of these scenarios, our algorithm converged to a width resulting in performance within 5% of optimal.

5.4.3 Our Algorithm in a Dynamic Environment

To test our adaptive algorithm under real-world dynamic conditions, we used publicly available traces of network traffic levels between hosts distributed over a wide area

during a two hour period [87]. For each host, the data values we use represent a one minute moving window average of network traffic every second, and we picked the 50 most heavily trafficked hosts as our simulated data sources. Traffic levels at these hosts ranged from 0 to $5.2 \cdot 10^6$ bytes per second. The simulated repository keeps an approximation of the traffic level for at most κ of the $n = 50$ sources, where κ is a parameter of the algorithm that is set to $\kappa = 50$ unless otherwise stated. Queries are executed at the repository every T_q seconds, computing either the MAX or SUM of traffic over 10 randomly selected sources. In all of our experiments, measurements of the overall cost per unit time $\mathcal{C}$ were taken after an initial warm-up period.

In our experiments, we consider refresh costs C_{vr} and C_{qr} that are intended to model network behavior under common consistency models and concurrency control schemes, although our algorithm handles arbitrary cost values. Usually, performing a remote read requires one request message and one response message, so $C_{qr} = 2$. If two-phase locking is used for transactional consistency of replicated bounds as mentioned in Section 4.2.2, then $C_{vr} = 4$ since two round-trips are required and thus $\rho = 2 \cdot \frac{4}{2} = 4$. Otherwise, if updates are simply sent to the repository, *e.g.*, in a multiversion or loosely consistent concurrency control scheme, then $C_{vr} = 1$ and $\rho = 2 \cdot \frac{1}{2} = 1$. Thus, most of our experiments consider the $\rho \in \{1, 4\}$ cases.

Figures 5.4 and 5.5 depict the exact value at one of the data sources for a short segment of a run, along with the replicated bound approximation as the value and bound change over time. For illustrative purposes we selected a portion of the run where a host became active after a period of inactivity. These figures illustrate the bound widths selected by our adaptive algorithm with parameters $\alpha = 1$, $\tau_0 = 0$, and $\tau_\infty = \infty$, when SUM queries are executed every second (*i.e.*, $T_q = 1$) and $\rho = 1$. Figure 5.4 uses average precision constraint $\delta_{avg} = 50K$, while Figure 5.5 uses average precision constraint $\delta_{avg} = 500K$.[1] When the average precision constraint is small, as in Figure 5.4, narrow bounds are favored. (To satisfy the precision constraints of SUM queries, the width of each of the 10 individual bounds being summed should be on the order of $\frac{\delta_{avg}}{10} = \frac{50K}{10} = 5K$.) When the average precision constraint is large, as in Figure 5.5, wide bounds (on the order of $\frac{\delta_{avg}}{10} = \frac{500K}{10} = 50K$) are favored.

[1] In the remainder of this section we abuse the abbreviation K for $\times 10^3$.

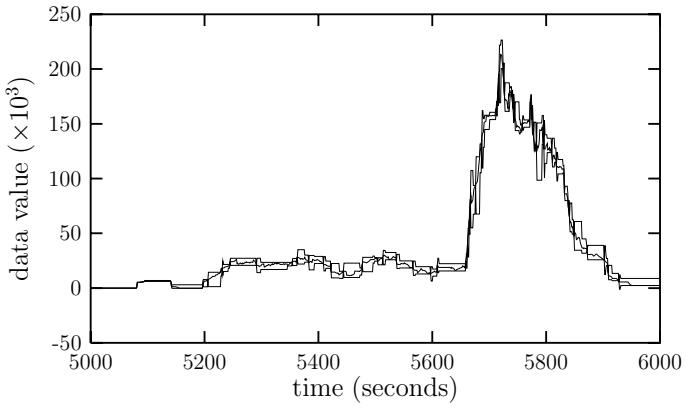


Figure 5.4: Source value and replicated bound over time for small precision constraints.

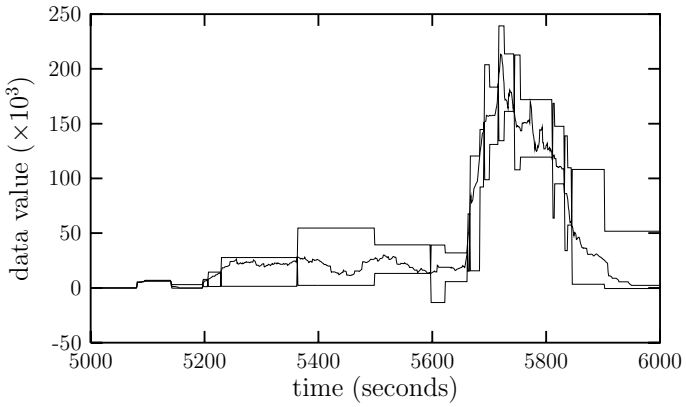


Figure 5.5: Source value and replicated bound over time for large precision constraints.

5.4.4 Setting Parameters

Next, we describe our experiments whose goal was to determine good values for the adaptivity parameter α and the lower and upper thresholds τ_0 and τ_∞. First, fixing $\tau_0 = 0$ and $\tau_\infty = \infty$ (meaning we never reset bound widths based on thresholds), we studied the effect of the adaptivity parameter α on performance. Figure 5.6 shows the results. We used SUM queries and varied α, considering several different settings for T_q, δ_{min}, δ_{max}, and ρ. In this and subsequent experiments, the y-axis cost $\mathcal{C}$ is the average for the entire run. All combinations of $T_q \in \{0.5, 1, 6\}$, $(\delta_{min}, \delta_{max}) \in \{(50K, 150K), (0, 100K)\}$, and $\rho \in \{1, 4\}$ are shown. From these experiments and

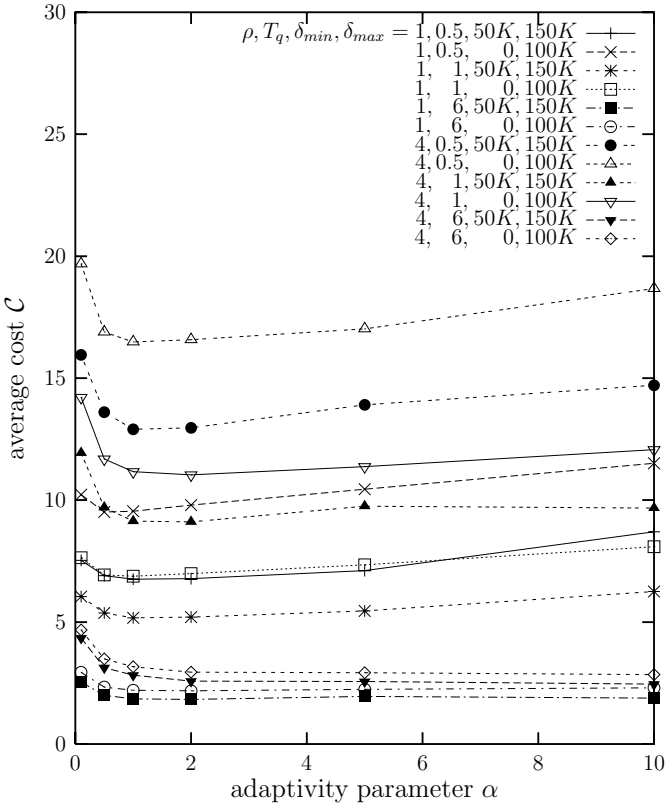


Figure 5.6: Effect of varying the adaptivity parameter α.

similar experiments using MAX queries, we determined that a good overall setting for α is 1. Recall from Section 5.2 that using $\alpha = 1$, the width W is doubled on value-initiated refreshes and halved on query-initiated refreshes.

We now address setting the lower threshold τ_0. Recall that the purpose of the lower threshold τ_0 is to force the bound width of an approximation to 0 when it becomes very small. A nonzero τ_0 parameter is necessary for queries that ask for exact answers, *i.e.*, have $\delta = 0$: with $\tau_0 = 0$, exact values would not be replicated, so such queries would always require source refreshes. It turns out that the performance under a workload with $\delta_{avg} = 0$ is not very sensitive to the value of τ_0, as long as $\tau_0 > 0$. However, setting τ_0 too large can adversely affect queries with small, nonzero precision constraints, since nonzero bounds of width below τ_0 are not permitted. Therefore, to accommodate queries with a variety of precision constraints, τ_0 should be set

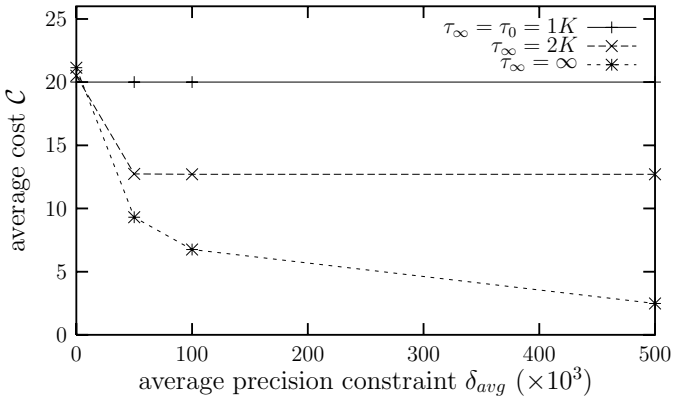


Figure 5.7: Performance of settings for τ_∞, query period $T_q = 0.5$.

to a small positive constant ϵ less than the smallest meaningful nonzero precision constraint. For our network data, differences in precision of $\epsilon = 1K$ are not very significant, so we set $\tau_0 = \epsilon = 1K$. This setting for τ_0 has only a small impact on queries even with moderately large precision constraints. For example, for precision constraints between $\delta_{min} = 5K$ and $\delta_{max} = 15K$, the performance degradation is less than 1% (for $T_q = 1$, $\tau_\infty = \infty$, and $\rho = 1$).

Having determined good values for α and τ_0, we now consider the upper threshold τ_∞. Setting τ_∞ to a small value improves performance for high-precision workloads since it eliminates replication of approximations that are not useful to queries. However, a small τ_∞ degrades the performance of low-precision workloads, *i.e.*, those having large precision constraints. To illustrate this tradeoff, in Figures 5.7, 5.8, and 5.9 we plot performance as a function of the average precision constraint δ_{avg}. Each graph corresponds to a different query period T_q and shows the performance using three different settings of τ_∞, holding all other parameters fixed: $\rho = 1$, $\Delta\delta = 0.5$, $\tau_0 = 1K$, and $\alpha = 1$. Workloads having $\delta_{avg} = 0$ perform best when $\tau_\infty = \tau_0$, which guarantees that all bounds are treated as having either no width ($W = 0$) or infinite width ($W = \infty$). That is, either the exact value is replicated or effectively no value is replicated at all. Since queries with $\delta = 0$ require exact precision, replicated bounds that are not exact are of no use. Note that whenever we set $\tau_\infty = \tau_0$, performance is independent of δ_{avg} as illustrated by the horizontal lines in Figures 5.7, 5.8, and 5.9.

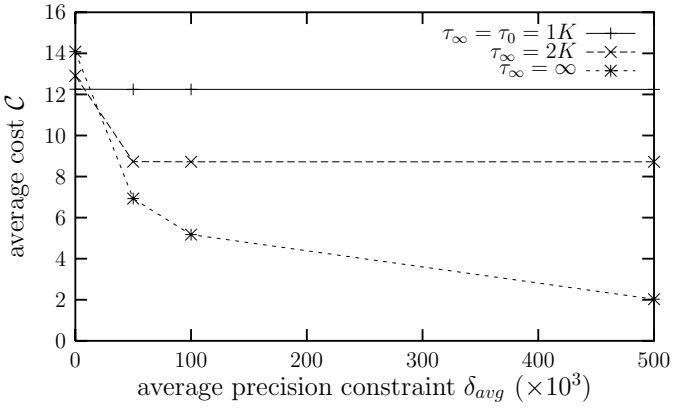


Figure 5.8: Performance of settings for τ_∞, query period $T_q = 1$.

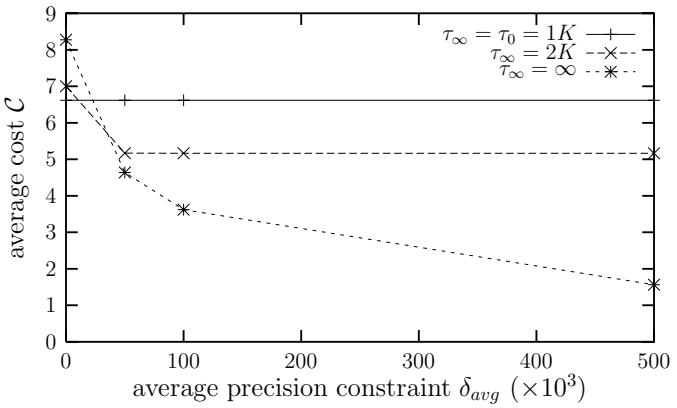


Figure 5.9: Performance of settings for τ_∞, query period $T_q = 2$.

For workloads having a range of different precision constraints, the upper threshold τ_∞ should be set to ∞.

Although these guidelines for setting the upper threshold τ_∞ apply to most types of queries including our SUM queries, there are exceptions. For example, values can be eliminated as candidates for the exact maximum based on bounds of finite, nonzero width (see Section 4.3.1). Therefore, for MAX queries, approximate values can be useful to replicate even when exact precision is required in all query answers. We have verified experimentally that for MAX queries, setting $\tau_\infty = \infty$ gives the best performance for all values of δ_{avg}, including $\delta_{avg} = 0$.

In summary, with parameter settings $\alpha = 1$, $\tau_0 = \epsilon$, and $\tau_\infty = \infty$, our algorithm

adaptively selects bounds that give the best possible performance under dynamically varying conditions. When $\Delta\delta = 0$, each query has the same precision constraint, so it is easier for the algorithm to discover the best precision for replicated bounds. On the other hand, if $\Delta\delta$ is large, each query has a different precision constraint, making it harder to find bound widths that work well across multiple queries. Fortunately, it turns out that the degradation in performance due to a wide distribution of precision constraints is small. We verified that the performance of our algorithm is not very sensitive to the precision constraint distribution for several different average precision constraints, while holding the other parameters fixed: $T_q = 1$, $\tau_0 = 1K$, $\tau_\infty = \infty$, and $\rho = 1$. When $\delta_{avg} = 100K$, the difference in performance between a workload with $\Delta\delta = 0$ and $\Delta\delta = 1$ is only 1.9%. When $\delta_{avg} = 10K$, the difference is 5.5%. When $\delta_{avg} = 5K$, the difference is less than 1%.

5.4.5 Unsuccessful Variations

We experimented with a number of variations to our algorithm that seemed intuitive but proved unsuccessful in practice: using uncentered bounds, using bounds that vary as functions of time, and adjusting bounds based on the refresh history. We report briefly on our experience with each variation.

A bound is *uncentered* if, at refresh time, the bound does not bound the exact value symmetrically. Thus, two width values are maintained instead of one: an upper width and a lower width. The source independently adjusts the upper and lower widths as follows. Each time a value-initiated refresh occurs due to the value exceeding the upper bound, then with probability $\min\{\rho, 1\}$ the upper width is increased. Conversely, when the value drops below the lower bound, with the same probability the lower width is widened. Whenever a query-initiated refresh occurs, with probability $\min\{\frac{1}{\rho}, 1\}$ both widths are decreased. In our experiments with both our synthetic random walk data and our real-world network monitoring data, the uncentered strategy performed worse than the centered strategy. However, in the case of synthetic *biased* random walk data, where values were much more likely to go up than down, using uncentered bounds improved performance slightly over using centered bounds.

A second unsuccessful variation is to use approximations that become more approximate over time. In Chapters 3 through Chap 5 our approximations are bounds $[L, H]$ whose endpoints are constant with respect to time. A more general approach is to make both L and H functions of time t. We ran experiments that showed that using bounds whose width increases with time proportionately to $t^{\frac{1}{2}}$ or $t^{\frac{1}{3}}$ resulted in worse performance than using constant bounds, both for the network monitoring data and unbiased synthetic random walk data. For biased random walk data (as described in the previous paragraph), the best bound functions turned out to be those having both endpoints increase linearly with time: $L(t) = k \cdot t$ and $H(t) = k \cdot t$, where the constant $k > 0$ is adjusted to match the average rate at which the data value increases. But, for general scenarios where the data does not predictably increase or decrease, constant bounds are preferred. Furthermore, constant bounds are much easier to index [63] than bounds that are functions of time. Finally, time-varying bounds can be tricky to implement, especially when an upper bound decreases or a lower bound increases with time, since an approximation can become invalid based on time alone.

A third variation we tried is to have the algorithm consider the past r refreshes when deciding how to adjust the bound. In this variation, the width is increased if the majority of the r most recent refreshes were value-initiated. Otherwise, the width is decreased. We also experimented with various techniques to weight recent refreshes within r more heavily. However, none of these schemes outperformed the algorithm presented here, which effectively sets $r = 1$ making it the most adaptive and simplest to implement.

5.4.6 Subsumption of Exact Replication

In this section we compare our algorithm against a state-of-the-art adaptive algorithm for deciding whether to maintain exact replicas, which we derive from the replication algorithm in [119]. In this algorithm, the number of requested reads r and writes w to each data value are counted. The replication strategy for every data value is reevaluated every x reads and/or writes to the value, *i.e.*, whenever $r + w \geq x$. At

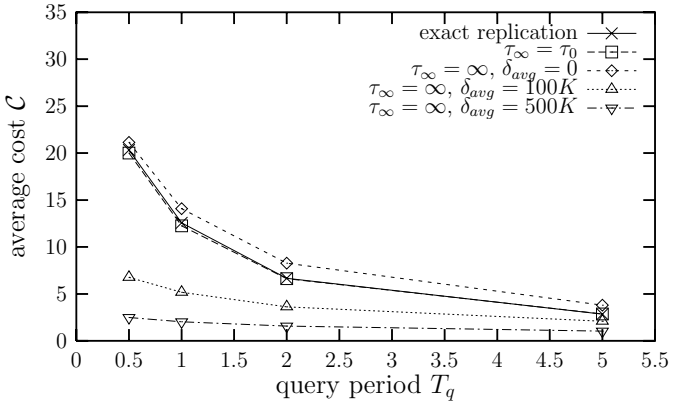


Figure 5.10: Comparison against exact replication, $\rho = 1$ and $\kappa = 50$.

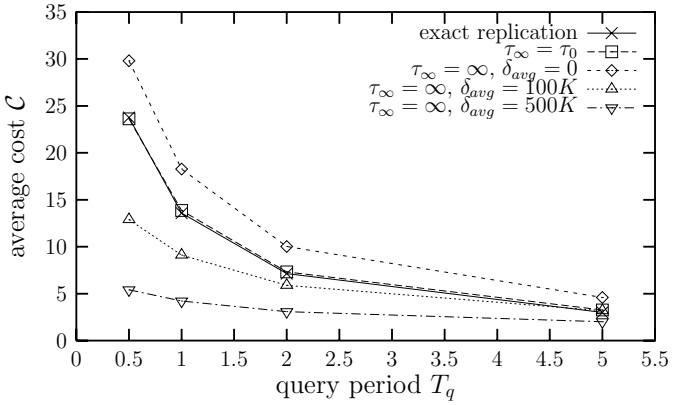


Figure 5.11: Comparison against exact replication, $\rho = 4$ and $\kappa = 50$.

reevaluation, the projected cost of not replicating, *i.e.*, the cost of performing r remote reads $\mathcal{C}_{nc} = r \cdot C_{qr}$ is computed. Similarly, the projected cost of replicating, *i.e.*, the cost of performing w remote writes, $\mathcal{C}_c = w \cdot C_{vr}$ is computed. The value is replicated if and only if $\mathcal{C}_c < \mathcal{C}_{nc}$. If the repository has limited space, values having the lowest cost difference $\mathcal{C}_{nc} - \mathcal{C}_c$ are evicted and the source is notified of the eviction. Under dynamic conditions, it has been shown that this adaptive exact replication strategy continually approaches the optimal strategy [119].

Figures 5.10, 5.11, 5.12, and 5.13 compare our algorithm against the exact replication algorithm of [119] for SUM queries executed every $T_q \in \{0.5, 1, 2, 5\}$ seconds.

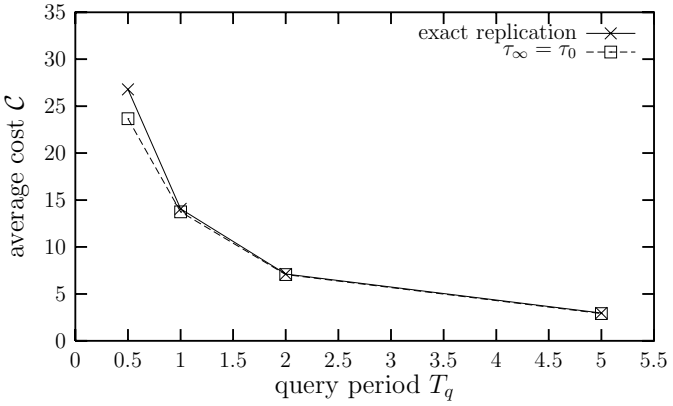


Figure 5.12: Comparison against exact replication, $\rho = 1$ and $\kappa = 20$.

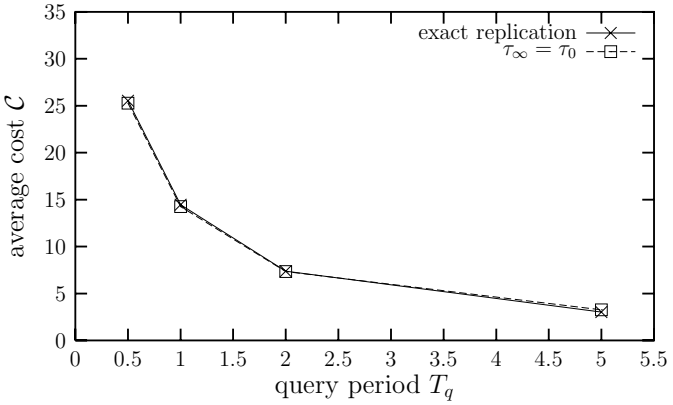


Figure 5.13: Comparison against exact replication, $\rho = 4$ and $\kappa = 20$.

For each run, we first determined the best setting for parameter x in the exact replication algorithm. Thus we changed the value of x, which varied from 3 to 45, between runs, whereas all of our own parameters remained fixed: $\alpha = 1$, $\tau_0 = 1K$, and $\tau_\infty \in \{1K, \infty\}$. Figures 5.10 and 5.11 show the results for a repository large enough to store all approximate values ($\kappa = 50$), with cost factor $\rho = 1$ and $\rho = 4$, respectively. Figures 5.12 and 5.13 show the results for a small repository of size $\kappa = 20$, again with $\rho = 1$ and $\rho = 4$ respectively.

The performance of our algorithm with $\tau_\infty = \tau_0$ almost precisely matches the exact replication algorithm under all workloads, repository sizes, and cost configurations tested. If we set $\tau_\infty = \infty$, our algorithm offers a significant performance improvement

for workloads not requiring exact precision, at the expense of a slight performance degradation for exact-precision workloads in the case of SUM queries, as shown in Figures 5.10 and 5.11. When MAX queries are used, our algorithm performs substantially better than exact replication because values can be eliminated as candidates for the exact maximum based on replicated bounds, as discussed in Section 5.4.4. When the repository size is limited, as in Figures 5.12 and 5.13, queries do not benefit much from nonzero precision constraints because inexact bounds tend to be evicted from the repository.

5.5 Related Work

The previous work most similar to that presented in this chapter is *Divergence Caching* [54], which also considers the problem of setting the precision of approximate values in a replication environment. In their setting, precision is based on number of updates to source values and not on the values themselves—the precision of a replicated approximation is inversely proportional to the number of updates since the last replica refresh. The Divergence Caching algorithm proposed in [54] works well in their environment, but does not generalize easily to the kinds of approximations we consider, which are based on the magnitude of source updates instead of their frequency.

No other work that we know of addresses precision setting of replicated approximate values while subsuming exact replication techniques. Work on *Moving Objects Databases* [118] considers setting precision of replicated approximations, but queries are not permitted to request exact values from sources so remote read costs are not taken into account. Conversely, in *Soft Caching* [61], updates to the exact source value are not considered, so value-initiated refreshes are not considered when setting precision.

5.6 Chapter Summary

In this chapter we presented a parameterized algorithm for adjusting the precision of approximate replicas not involved in continuous queries. Our algorithm adjusts precision levels adaptively to achieve the best performance for a workload of one-time queries as data values, precision requirements, or workload vary. As in the previous chapter we considered interval approximations to numeric values but our ideas can be extended to other kinds of data and approximations. Our algorithm strictly generalizes previous adaptive replication algorithms for exact copies: we can set parameters to require that all approximations be exact, in which case our algorithm dynamically chooses whether or not to replicate each data value.

We implemented our algorithm and tested it on synthetic and real-world data. A number of experimental results were reported, showing the effectiveness of our algorithm at maximizing performance, and also showing that in the special case of exact replication our algorithm performs as well as previous algorithms. In cases where bounded imprecision is acceptable, our algorithm easily outperforms previous algorithms for exact replication.

Chapter 6

Visual User Interfaces for Approximate Replication Systems

6.1 Introduction

In this chapter we shift our focus to user interface issues related to approximate replication. In particular, we study issues that arise with the use of our techniques in conjunction with automated *data visualization*, a common method of end-user data analysis in which a display consisting of charts, graphs, etc. is produced to depict the data in graphical form, *e.g.*, [6, 12, 30, 73, 104, 122]. Data visualization is used to support data analysis in a wide variety of domains including science, engineering, business, and others. However, people in all fields tend to be wary of relying heavily on visualization because of the potential for graphical depictions to mislead users and encourage improper interpretation [55].

The presence of *uncertainty* in data being visualized is one potential cause of misleading depictions. Most work on visualization to date has assumed that the underlying data entries are either exact values, or statistical expected values derived from a distribution of possible values. However, the very different form of uncertainty that arises from accessing or querying approximately replicated values has largely been left unaddressed. When a numeric bound is used as an approximate replica of a data object, as discussed in Chapters 3–5, although it is known for certain that the value lies

somewhere inside the bound interval, it is not known where. Furthermore there may be no known probability distribution of possible values within the interval. We refer to this form of uncertainty as *bounded uncertainty*, and differentiate it from *statistical uncertainty*, which is typically (although not always) characterized by a potentially infinite distribution of possible values with a single peak indicating the most likely estimate.[1] Bounded uncertainty also occurs, for example, when a replica of a numeric data object is refreshed periodically, and there is a known and finite maximum rate of change for the master value. Another scenario in which bounded uncertainty arises is the compression of numeric time-series data using piecewise constant approximations with error guarantees, as in [20, 69].

Pang et al. [86] argue that uncertainty should be presented along with data in visualization applications, and we develop this approach for bounded uncertainty. After discussing traditional techniques for showing statistical uncertainty such as drawing error bars, [86] proposes an extensive suite of techniques for conveying uncertainty in 3D geometry visualization applications. Many of these techniques can be adapted to data visualization scenarios that involve abstract charts and graphs. However, techniques for conveying statistical uncertainty tend to be misleading when used for bounded uncertainty for two reasons. First, users have been trained to interpret them as probabilistic bounds on an unbounded distribution of possible values. In contrast, bounded uncertainty is characterized by nonprobabilistic endpoints. Second, since error bars and related techniques for conveying statistical uncertainty are typically used in conjunction with an estimated exact value, the existence of a single most likely value is strongly implied. There is typically no most likely value in the case of bounded uncertainty.

We believe that visualizations should clearly differentiate between the two forms of uncertainty, making it obvious whether the uncertainty is statistical or bounded, in addition to conveying the degree of uncertainty. To convey statistical uncertainty, it is appropriate to display the most likely value along with error bars or other glyphs as in [86]. To convey bounded uncertainty, as in our replication environments that

[1] A report by the US Department of Commerce National Institute of Standards and Technology (NIST) [105] identifies statistical and bounded uncertainty as two predominant forms in which uncertainty in data tends to occur.

provide guaranteed bounds for numeric query answers, we advocate a systematic technique based on widening the boundaries and positions of graphical elements and rendering the uncertain region in fuzzy ink, or using a lighter gradation of shading. In this chapter we show how to apply this technique, which we call *ambiguation*, to common displays of abstract charts and graphs. Interestingly, it is not always possible to show the exact degree of uncertainty, and in some cases it can only be displayed approximately. We specify an algorithm that approximates the degree of uncertainty to make it displayable while minimizing the overall loss in accuracy.

We also consider new issues raised when visualization is performed online, over the results of continuous queries evaluated on approximate replicas. In this scenario, data and uncertainty levels can change dynamically, and it is easy for users to falsely interpret sudden changes in the uncertainty bounds as changes in the underlying data. We discuss masking sudden jumps in uncertainty when they do not correspond to significant data changes, to avoid inappropriately drawing the user's attention.

Finally we consider the central feature of our approximate replication techniques in Chapters 3–5, which is that applications can control the degree of uncertainty. While it can be beneficial to pass on this control over uncertainty levels to end-users, naive interfaces may not effectively expose the tradeoff involved. As a result, users may unknowingly abuse this power. We propose ways to offer users control over uncertainty levels that encourage judicious use of the control mechanism.

6.1.1 Chapter Outline

The remainder of this chapter is structured as follows. In Section 6.2 we describe our systematic approach to conveying the presence, form, and degree of uncertainty in data. Then, we address misleading sudden jumps in Section 6.3. In Section 6.4 we discuss user-controlled uncertainty tuning for interactive visualization over continuous query results in approximate replication environments. Finally, we provide a discussion of related work in Section 6.5 and a summary of this chapter in Section 6.6.

6.2 Representing Uncertain Data Visually

In most abstract charts and graphs, data values are graphically encoded either in the positions of graphical elements, as in a scatterplot, or in the extent (size) of elements along one or more dimensions, as in a bar chart. When the underlying data is uncertain, we believe it is appropriate to clearly indicate not only the presence and degree but also the form of uncertainty. As described in Section 6.1, statistical and bounded uncertainty encode two dramatically different distributions of potential values, so we advocate two alternative methods for conveying uncertainty in the positions or extents of graphical representations of data: *error bars* for statistical uncertainty and *ambiguation* for bounded uncertainty. We begin by describing these general techniques and then show how they can be applied to some common types of charts and graphs.

6.2.1 Error Bars

Error bars and their variants (quantile plots, etc.) have been well studied as a suitable means to convey statistical uncertainty [26, 107, 108]. Typically, a normal distribution is assumed. For each uncertain data value to be represented visually, the idea is to use the normal display technique to render the expected value in place of the unknown exact value. Error bars are then added to indicate uncertainty in the position or boundary location in proportion to the size of an associated confidence interval. Some standard uses of error bars are illustrated in the upper left quadrant of Figure 6.1 (we explain the rest of the figure below in Section 6.2.4). When uncertainty occurs in bounded rather than statistical form, it is important to avoid the use of error bars since the accepted interpretation implies a potentially unbounded distribution extending beyond the error bars. Even worse, rendering an exact estimate using the normal display technique strongly implies the existence of a most likely value, but in bounded uncertainty no most likely value can be assumed.

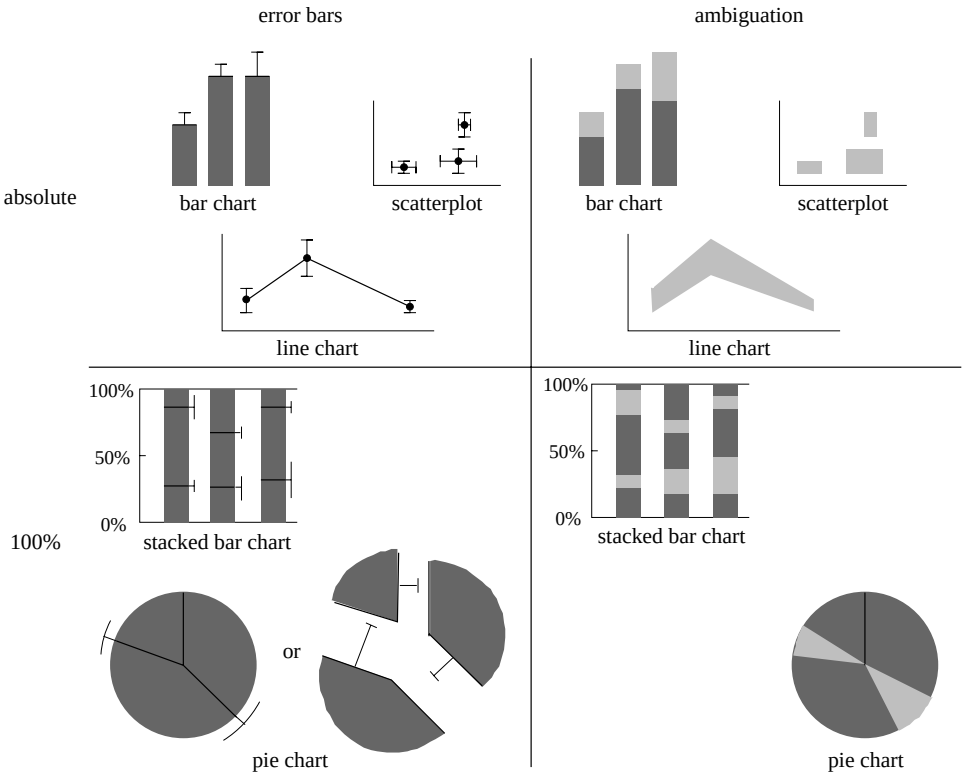


Figure 6.1: Error bars and ambiguation applied to some common chart types.

6.2.2 Ambiguation

To convey the presence and degree of bounded uncertainty, we propose the use of a new technique we call ambiguation. The main idea behind ambiguation when uncertain data is encoded in the extent of a graphical element is to widen the boundary to suggest a range of possible boundary locations and therefore a range of possible extents. The ambiguous region between possible boundaries can be drawn as graphical fuzz using light shading or dithering, giving an effect that resembles ink smearing. A straightforward application of this technique is illustrated in the ambiguated bar chart in the upper right quadrant of Figure 6.1. To indicate positional uncertainty, rather than drawing a crisp representation of the graphical element at a particular position, the representation is elongated in one or more directions and drawn using fuzz. A simple application of this technique is illustrated in the ambiguated scatterplot in the upper right quadrant of Figure 6.1.

Other variations of boundary or position ambiguation may be possible, but the necessary feature is that no particular estimate or most likely value should be indicated. Rather, the entire range of possible values for the boundary or position of the graphical element should be presented with equal weight. This key characteristic is in contrast with error bars and other approaches such as fuzzygrams and gradient range symbols [48] that emphasize a known probability distribution over data values.

6.2.3 Discussion

The complementary use of error bars and ambiguation makes the presence, degree, and form of uncertainty clear. First, these techniques make it easy to identify the specific data values that are uncertain by suggesting imprecision in the graphical property (position or boundary location) in which the values are encoded. For bounded uncertainty, the position or boundary is made ambiguous using fuzzy ink, and for statistical uncertainty, error bars are added to visually suggest the possibility of a shift in position or boundary location. Second, these techniques allow the degree of uncertainty to be read in a straightforward manner using the same scale used to interpret the data itself. Finally, the use of two visually distinct techniques makes it clear which of the two forms of uncertainty is present, and each technique conveys the properties of the form of uncertainty it represents.

Ambiguation and error bars work well when data is encoded as the position or extent of graphical elements. Coping with displays that use other graphical attributes such as color and texture to encode data is left as a topic for future work. In the absence of analogous techniques for other graphical attributes, when uncertainty is present it is desirable to only use charts and graphs that encode data using position and extent alone so the presence, degree, and form of uncertainty can be clearly and unambiguously depicted.

6.2.4 Application to Common Chart Types

Figure 6.1 illustrates how error bars and ambiguation can be applied to some common chart types. While these techniques are general and can be applied to a broad range

of displays that use position and extent to encode data, we focus on abstract charts and graphs, which can be classified into two categories: *absolute displays* and *100% displays*. In absolute displays, each data value is given a graphical representation whose extent or position is plotted on an absolute scale. Examples of absolute displays include simple bar charts (which encode data in the upper boundaries of bars), scatterplots (which encode data in the positions of points), and line graphs (which encode data in the positions of points and lines). It is generally straightforward to add error bars or apply ambiguation to boundaries and positions in absolute displays such as those displayed in the top half of Figure 6.1.[2]

In 100% displays, the scale ranges from 0% to 100%, and n values $V_1, V_2, \ldots, V_n$ are plotted on this relative scale. Each value V_i is plotted as a graphical element whose size is proportional to the fraction $\frac{V_i}{\sum_{1 \leq j \leq n} V_j}$ of the total over all n values. Examples of 100% displays include stacked bar charts and pie charts. Indicating uncertainty in 100% displays is more challenging than doing so in absolute displays. In 100% displays, the graphical elements usually contact each other directly, so the boundary between two elements indicates the difference between them in terms of relative contribution to the total. To inform the user of statistical uncertainty in the locations of these boundaries, error bars can be drawn adjacent to the boundaries. Alternatively, for pie charts the wedges can be separated, leaving space for error bars extending directly from the boundaries between wedges. The lower left quadrant of Figure 6.1 illustrates these techniques. Bounded uncertainty can be indicated by inserting an ambiguous region of fuzzy ink between each pair of elements whose shared boundary is uncertain, as illustrated in the lower right quadrant of Figure 6.1. In the example pie chart, the fractional contribution to the total from the lower-left wedge may be as high as 50%, due to the indicated range of possible locations for both boundaries shared with neighboring wedges. It turns out that determining the sizes to use for the fuzzy and solid regions in an ambiguated 100% display is not trivial

[2]Displays such as stacked bar charts that are not normalized to sum to 100% are problematic when used in conjunction with these uncertainty indicators because the interpretation can be ambiguous. For example, in an absolute stacked bar chart, an error bar or a fuzzy region appearing at the top of the stack can be interpreted either as uncertainty in the topmost element or as uncertainty in the overall height of the stack (the sum over all elements).

because each region of fuzz shares a border with two solid data regions. We address this challenge next.

Ambiguation in 100% Displays

Here we consider how to draw an ambiguated pie chart (the formulation for a stacked bar chart is similar). Suppose the data for the pie chart consists of an ordered list of n objects $O_0, O_1, \ldots, O_{n-1}$ whose values are known to lie inside the intervals $[L_0, H_0], [L_1, H_1], \ldots, [L_{n-1}, H_{n-1}]$, respectively. As a first step, the absolute uncertainty intervals need to be converted into relative ones that indicate the smallest and largest possible fraction of the chart covered by each data object:

$$L_i^r = \frac{2\pi \cdot L_i}{\sum_{j=0}^{n-1} H_j - H_i + L_i} \qquad H_i^r = \frac{2\pi \cdot H_i}{\sum_{j=0}^{n-1} L_j - L_i + H_i}$$

The smallest possible fraction L_i^r occurs when the value of O_i is as low as possible, *i.e.*, equal to L_i, and the values of all other objects $O_j \neq O_i$ are as high as possible, *i.e.*, equal to H_j. The rationale for H_i^r is symmetric.

Ideally, an allocation of fuzzy and solid ink that conveys the uncertainty exactly could be found, so that each data object O_i has a corresponding solid pie wedge of arc length L_i^r (in radians) and two adjacent fuzzy wedges of total arc length $H_i^r - L_i^r$. For example, suppose we wish to draw a pie chart for two data objects, each with values in the interval $[1, 2]$, and thus relative contributions of between $\frac{1}{3}$ and $\frac{2}{3}$ each. A simple chart with two solid wedges of arc length $\frac{2\pi}{3}$ each plus a fuzzy wedge also of arc length $\frac{2\pi}{3}$ achieves the ideal of conveying exactly the uncertainty intervals present in the data.

Unfortunately, due to the nature of 100% displays, this ideal is not always achievable. In some cases it is not possible to convey the exact uncertainty intervals. For example, suppose we wish to draw a pie chart for three data objects with identical intervals of $[1, 2]$, or relative contributions of between $\frac{1}{5}$ and $\frac{1}{2}$ each. To indicate the smallest possible relative contribution of each data object, we need three solid wedges of arc length $\frac{2\pi}{5}$ each. The three gaps between the wedges will be filled with fuzz. We denote the arc lengths of the three fuzzy regions as F_0, F_1, and F_2. No matter

how we arrange the solid wedges, the combined arc length of the fuzzy regions is $F_0 + F_1 + F_2 = 2\pi - 3 \cdot \frac{2\pi}{5} = \frac{4\pi}{5}$. To convey that the maximum possible relative contribution of each data object is $\frac{1}{2}$, we need to arrange the three solid wedges, with fuzz in between, such that each solid wedge taken together with the two adjacent fuzzy regions spans a total arc length of π (half circle). Therefore, we require that $F_0 + \frac{2\pi}{5} + F_1 = F_1 + \frac{2\pi}{5} + F_2 = F_2 + \frac{2\pi}{5} + F_0 = \pi$. We have defined a system of four equations with three unknowns (F_0, F_1, and F_2) that is overconstrained and has no solution. Therefore, an arrangement of solid wedges that conveys the exact uncertainty intervals that occur in the data does not exist in this case.

The only way to draw an ambiguated pie chart in such cases is to approximate the level of uncertainty that occurs in the data, which is undesirable but necessary. In general, consider the task of creating a pie chart with ambiguation for a data set consisting of n objects $O_0, O_1, \ldots, O_{n-1}$. Let $S_i \geq 0$ be the new arc length of the solid portion of the display for O_i, and let $F_i \geq 0$ be the arc length of the fuzzy region between the solid region for the two objects O_i and $O_{(i+1) \bmod n}$. An optimization problem arises with the goal of minimizing the total amount of fuzz without giving false information, *i.e.*, without drawing a chart indicating uncertainty intervals that do not contain the actual uncertainty intervals intrinsic in the data: Minimize $\sum_{i=0}^{n-1} F_i$ such that $\sum_{i=0}^{n-1} S_i + \sum_{i=0}^{n-1} F_i = 2\pi$ and for all i: $S_i \leq L_i^r$ and $F_{(i-1) \bmod n} + S_i + F_i \geq H_i^r$.

The objective function minimizes the total amount of fuzz, which in turn minimizes the overall loss in accuracy due to the use of approximation. The first constraint requires that all of the solid and fuzzy wedges placed together create a full circle. The second constraint ensures that, for each object, the arc length of the solid region is no larger than the minimum relative contribution L_i^r of the object's value to the total. The third constraint ensures that, for each object, the combined arc length of the solid region taken together with the two adjacent fuzzy regions is no smaller than the maximum relative contribution H_i^r of the object's value to the total. The second and third constraints together ensure that the uncertainty intervals implied by the chart contain the actual uncertainty intervals they approximate.

We have implemented an ambiguated pie chart renderer that invokes a publicly available linear program solver to solve this optimization problem and determine the

best possible pie chart layout to minimize the loss in accuracy due to displaying approximate uncertainty intervals. It runs in under 10 milliseconds on a modest workstation for data sets of 25 objects, which is large for a 100% chart. It is therefore suitable for execution as a part of an interactive rendering cycle. In certain extreme cases where a data set exhibits a great deal of uncertainty, the linear program has no solution, and it is impossible to generate an ambiguated 100% chart, even by approximating the uncertainty intervals. This problem can sometimes be resolved by reordering the wedges, but this method may cause a disconcerting effect in dynamic displays of changing data. Instead, we advocate using a special icon when no pie chart can be drawn to indicate extreme uncertainty. The user can react by requesting a decrease in uncertainty to make the chart displayable, using the interface proposed below in Section 6.4.

6.3 Avoiding Misleading Sudden Jumps

When the graphical display is refreshed due to a change in the underlying data, sudden jumps in the displayed data will tend to draw the user's attention. This characteristic is usually appropriate when the jump corresponds to a drastic change in the data. However, consider the case where the source of uncertainty is an approximate replication system, as discussed in Chapters 3–5. In approximate replication, systems considerations, rather than semantic events, trigger the source to refresh the replicated interval, so sudden jumps in the graphical display of data and uncertainty level may not correspond to significant changes in the underlying data. Moreover, the absence of jumps may not rule out changes. Therefore, in visualization applications that access approximate data via approximate replication protocols, it is desirable to mask sudden jumps in the data or uncertainty level that would inappropriately draw the user's attention. Sudden jumps can be masked by smoothly animating the transitions between old and new data and uncertainty values at a slow enough rate to not be overly distracting.

6.4 Controlling the Degree of Uncertainty

In environments where the data being visualized is obtained from an approximate replication system, there is an opportunity to exert control over the uncertainty levels at a per-object or per-query granularity. In the *Precision Fixed/Maximize Performance* scenario covered in Chapters 3–5, a decrease in the uncertainty of some data objects or query results is offset by an increase in communication resource utilization. End-users may not be aware of this tradeoff, and consequently they may unknowingly abuse the power to control the uncertainty level by requesting unnecessarily precise representations and incurring excessive cost behind the scenes. Therefore, it may be desirable for the visualization system to mediate control over the uncertainty levels.

One way to perform this mediation is for the visualization system to maintain default uncertainty levels[3] that are either uniform, lower for graphical elements near the center of the screen, or lower for elements that have remained on the screen for a long time. In some situations in which bounded uncertainty is displayed, it may also be appropriate to require that uncertain regions do not overlap in the dimension(s) of interest, making the relative order of data values always discernible. At any point, the user can override the default uncertainty levels by clicking on an area of interest, causing that area to "come into focus" via a decrease in uncertainty. Then, the visualization system should gradually return the uncertainty levels to their default state.

When network bandwidth is plentiful, there is no limit to how much the overall uncertainty can be reduced. However, lower uncertainty may incur a higher communication cost. It may be important to convey the cost to the user via a network traffic status indicator, for example, so that the increase in traffic resulting from requesting lower uncertainty levels is indicated visually. Noticing the increase in traffic, the user could click on the network indicator to reduce traffic again by affecting a mild increase

[3]Users of approximate replication systems might wish to modify the default uncertainty level of a particular data object by specifying the lower and upper endpoints of the uncertainty interval, causing the display of that object to remain static until the data value moves beyond one of the endpoints. This feature can serve to alert users when a data value exceeds a critical threshold.

in uncertainty across the board. If the network traffic indicator does not provide adequate incentive for the user to be sparing when lowering uncertainty levels, it may be appropriate to have uncertainty levels continually increase by default. This property would force the user to click periodically to maintain data in sharp focus, requiring effort commensurate with the amount of work required of the network infrastructure, thereby implicitly communicating the cost to the user. Furthermore, if the user abandons the visualization for, say, a coffee break, without closing the application, the display would eventually become entirely out of focus, incurring no network costs while the visualization is not in use.

6.5 Related Work

In certain visualization scenarios, data may be unavailable for display or even purposefully omitted for a variety of possible reasons, giving rise to uncertainty. The importance of visually informing the user of the absence of data has been identified [121] and techniques for doing so have been proposed in, *e.g.*, Clouds [8, 49] and Restorer [109]. We focused on a different type of uncertainty where all the data is present but precise values are not known.

Numerous ways to convey the degree of uncertainty in data using overlayed annotations and glyphs have been proposed, as in, *e.g.*, [86]. Another approach is to make the positions of grid lines used for positional reference ambiguous [21]. Uncertainty can also be indicated by adjusting the color, hue, transparency, etc. of graphical features as in, *e.g.*, [29, 74, 111]. Some techniques for conveying uncertainty by widening the boundaries of graphical elements have also been proposed. For example, in [116], the degree of uncertainty in the angle of rotation of vectors is encoded in the width of the vector arrows. Also, [86] proposes varying the thickness of three-dimensional surfaces to indicate the degree of uncertainty.

To our knowledge, however, none have focused on accurately and unambiguously conveying not only the presence and degree but also the form of uncertainty in data, as we did in this chapter. We also believe that this work is the first to establish systematic methods for conveying bounded uncertainty by widening the boundaries

and positions of graphical elements in abstract charts and graphs. The approach in [39] for displaying cluster densities gives a visual appearance similar to our ambiguated line charts (discussed later) but serves a different purpose.

Our work also addressed control over uncertainty levels. Interfaces for controlling uncertainty levels were proposed in [49], but that work does not address ways to make the user cognizant of tradeoffs between decreased uncertainty and increased resource utilization.

6.6 Chapter Summary

Visualization is a powerful way to facilitate data analysis, but it is crucial that visualization systems explicitly convey the presence, nature, and degree of uncertainty to users. Otherwise, there is a danger that data will be falsely interpreted, potentially leading to inaccurate conclusions.

In this chapter we argued that error bars or similar techniques designed to convey the presence and degree of statistical uncertainty should not be used when visualizing data that exhibits bounded uncertainty, which has different properties. We described a technique for conveying bounded uncertainty in visualizations and showed how it can be applied systematically to common displays of abstract charts and graphs. Interestingly, it is not always possible to show the exact degree of uncertainty, and in some cases it can only be displayed approximately. We specified an algorithm that approximates the degree of uncertainty to make it displayable while minimizing the overall loss in accuracy. In addition, we proposed interfaces that offer control of uncertainty levels to end-users of approximate replication systems while encouraging judicious use of the precision-performance tradeoff.

Chapter 7

Summary and Future Work

This dissertation studied the problem of making efficient use of communication resources in approximate replication environments. We framed our analysis in terms of a two-dimensional space with axes denoting system *performance* (a measure of resource utilization) and replica *precision* (a measure of the degree of replica synchronization), and observed that there is a fundamental and unavoidable tradeoff between precision and performance. Although this tradeoff seems uncircumventable, we noted that a push-based approach to replica synchronization offers the opportunity for the best precision-performance curves, *i.e.*, the best precision for a certain performance level and, conversely, the best performance for a certain precision level. These initial observations led us to propose and study two complementary methods for working with the precision-performance tradeoff to achieve efficient resource utilization for replica synchronization:

1. Push-based approximate replication with the goal of maximizing replica precision in the presence of constraints on system performance.

2. Push-based approximate replication with the goal of maximizing system performance in the presence of constraints on replica precision.

We began by proposing a push-based approximate replication technique for the *Performance Fixed/Maximize Precision* scenario in Chapter 2. To cope with constraints placed on communication performance by users, applications, or the environment, sources prioritize refreshes of replicas maintained at the central repository based on precision considerations, and only send refreshes whose priority exceeds a certain threshold. A distributed algorithm adjusts the refresh priority threshold at each source adaptively to attempt to achieve the highest possible overall replica precision while adhering to constraints on communication performance. We showed that our push-based technique, which relies heavily on source cooperation, achieves significantly better precision than the best known pull-based algorithm, supporting our hypothesis of the superiority of push-based replication over pull-based approaches in terms of achieving better precision with the same performance.

Next, in Chapters 3–5 we studied the inverse scenario of *Precision Fixed/Maximize Performance*, which is of interest when network resources are not severely limited but communication incurs some cost. In our approach, users submit queries over the replicated data at the repository together with custom precision requirements, and answers of sufficient precision are produced while incurring minimal communication cost. In Chapter 3 we focused on continuous queries with precision requirements, and proposed an algorithm for continuously adjusting the precision of individual approximate replicas adaptively so as to converge to the minimum cost configuration. We also reported experimental results from a real-world network traffic monitoring system based on continuous queries over approximate replicas.

We then turned our attention to unanticipated one-time queries with precision requirements. Due to the ad-hoc nature of one-time queries, the data replicas in the repository may not be of sufficient precision to meet the query's precision requirement. In Chapter 4 we introduced algorithms that access a minimum-cost subset of master data objects from their remote sources to guarantee adequate query result precision.

In Chapter 5 we studied the important problem of deciding what precision levels to use for replicas not involved in continuous queries but subject to intermittent accesses by one-time queries. This problem generalizes the previously studied problem of deciding whether to perform exact replication of individual data objects. We proposed

an adaptive algorithm for setting replica precision with the goal of maximizing overall communication performance given a workload of one-time queries. In an empirical study we compared our algorithm with a prior algorithm that addresses the less general exact replication problem, and we showed that our algorithm subsumes it in performance.

Finally, in Chapter 6 we proposed techniques for adapting some standard data visualization methods to handle the types of approximate values produced by our replication techniques, with the goal of not misleading users regarding data precision. We also discussed the possibility that end-users be permitted to specify and adjust precision requirements for continuous queries over approximate replicas directly in the visualization interface, and proposed some potential means of motivating users to request only as much precision as they need so as not to incur excessive cost.

The global contribution of this dissertation is to steer research on replication of volatile data toward principled techniques for combined management of communication resources and data precision. The volume of automatically-generated data from sensors and other electronic measurement devices is increasing at a dramatic rate. This trend, coupled with the tendency toward distributed computing architectures, drives the need for efficient techniques for remote monitoring and querying of distributed and rapidly-changing data. Meanwhile, in many emerging applications that monitor and query volatile data, exact precision is not required. Instead, in many scenarios *approximate replication* is sufficient, and is much less costly to achieve than exact replication.

This dissertation has examined the approximate replication problem formally and from a number of different angles, and has also provided extensive experimental validation of the techniques developed. Our hope is that the applicability of this work spans a broad spectrum of real-world scenarios. To further test the usefulness and benefits of our techniques, we plan to expand our prototype network monitoring implementation to include a more extensive suite of traffic monitoring and querying facilities, with an emphasis on flexibility and ad-hoc querying. To realize our goal of developing a very general framework for resource-efficient network traffic monitoring, we will need to expand our work on approximate replication in a number of ways.

A general list of some of the directions in which we envision expanding our work is provided below.

7.1 Future Work

- **Automatically choosing between our complementary approaches.** The choice of which of the complementary approaches of *Performance Fixed/Maximize Precision* and *Precision Fixed/Maximize Performance* is more appropriate in a given scenario depends on characteristics of the environment and application. An interesting topic for future work is to consider policies for automatically choosing or switching between the two approaches, possibly at a per-source granularity.
- **Expanding our techniques to handle delta encoding.** If data objects are large, we may want refresh messages to encode the difference (delta) between the current source copy and the out-of-date replica, rather than sending the entire object. Incorporating such a technique, *e.g.*, as used in [70, 81], into our approach would require some significant modifications because the refresh cost may increase with the number of updates to the master source copy.
- **Considering batching multiple refreshes together.** In some environments it may be appropriate to amortize network bandwidth by packaging several data objects into the same message for refreshing. Doing so will cause some refreshes to be delayed artificially while the source waits for other refreshes to accumulate. It would be interesting to explore the tradeoff between packaging multiple refresh messages together to save bandwidth versus the increased divergence or necessary increase in the latency tolerance λ (Chapter 3) resulting from delaying refreshes.

7.1.1 Performance Fixed/Maximize Precision

We now discuss some avenues for future work that pertain specifically to the *Performance Fixed/Maximize Precision* aspect of this dissertation studied in Chapter 2.

- **Studying applications with mutual consistency requirements.** In some applications we may need to maintain mutual consistency requirements among objects being replicated [110], which would constrain the order in which refreshes could be performed.
- **Considering priority functions based on an extended history window.** In our best-effort synchronization approach described in Chapter 2, the refresh priority of an object is based solely on the updates that have occurred since the last refresh. Although our experiments indicate that this approach works quite well, it might be interesting to consider priority functions based on a longer history period, to trade adaptiveness and reduced state for possibly more reliable predictions of future behavior.
- **Studying priority-based refresh scheduling in the presence of nonuniform refresh costs.** We can extend our techniques to environments where the cost to refresh objects is not uniform, possibly because they have different sizes. Accounting for nonuniform cost in the priority function is a simple matter of extending the weight to include a factor inversely proportional to cost. However, then the highest priority object could have high cost and potentially require more resources than are currently available, while a lower priority object could be refreshed. It is not obvious how best to manage bandwidth usage in a dynamic environment when objects have nonuniform cost.

7.1.2 Precision Fixed/Maximize Performance

We divide the future directions for the *Precision Fixed/Maximize Performance* component of this dissertation (Chapters 3, 4, and 5) into three categories: additional functionality, choosing tuples to refresh for one-time queries, and improving performance.

Additional Functionality

- **Expanding the class of aggregation queries we consider.** We have devised algorithms for other aggregation functions, such as MEDIAN (for which

preliminary results are reported in [38]) and TOP-k (continuous TOP-k queries are studied in [9]). In addition, we would like to extend our results to handle grouping on bounded values, enabling GROUP-BY and COUNT UNIQUE queries. Nested aggregation functions such as MAX(AVG) have been studied in [62].

- **Looking beyond aggregation queries.** We believe that our ideas can be expanded to encompass other types of relational and non-relational queries having different precision constraints. In our running example (Section 4.1.2), suppose we wish to find the lowest latency path in the network from node N_i to node N_j. A precision constraint might require that the value corresponding to the answer returned by the system (*i.e.*, the latency of the selected path) is within some distance from the value of the precise best answer. Initial complexity results for this problem [37] indicate that there is little hope of finding an optimal solution, so approaches based on heuristics are probably necessary.

- **Studying adaptive precision-setting techniques for non-numeric data.** Our overall approach setting the precision of approximate replicas in the presence of one-time queries can be applied with a broad variety of data types. Tuning the details of our approach for nonnumeric data types such as Web pages may be a worthwhile undertaking.

- **Allowing users to express relative instead of absolute precision constraints.** A relative precision constraint might be expressed as a constant $P \geq 0$ that denotes an absolute precision constraint of $\delta = 2 \cdot A \cdot P$, where A is the actual answer. For continuous queries, we can meet a relative precision constraint by continually adjusting δ (see Section 3.4.6) to match the current upper answer bound H. Relative precision constraints in the context of one-time queries over bounded values have been studied in [62].

- **Considering applying our ideas to *multi-level* replication systems, where each data object resides on one source and there is a hierarchy of data repositories.** Refreshes would then occur between a repository and the repositories or sources one level below, with a possible cascading effect.

Initial work in this area includes [20] and [100].

- **Studying precision setting in *symmetric* replication systems.** It may be interesting to investigate adaptive precision setting in symmetric (peer-to-peer) replication architectures, building on work on approximate replication in such environments, *e.g.*, [125].

Choosing Objects to Refresh for One-Time Queries

- **Adapting our** CHOOSE_REFRESH **algorithms to take refresh batching into account.** If multiple query-initiated refreshes are sent to the same source, the overall cost may be less than the sum of the individual costs. We would like to adapt our CHOOSE_REFRESH algorithms to take into account such cases where refreshing one tuple reduces the cost of refreshing other tuples. In fact, the same adaptation may help us develop CHOOSE_REFRESH algorithms for queries involving join and group-by expressions. In both of these cases, refreshing a tuple for one purpose (one group or joined tuple) may reduce the subsequent cost for another purpose (group or joined tuple).
- **Considering iterative** CHOOSE_REFRESH **algorithms.** Rather than choosing a set of tuples in advance that guarantees adequate precision regardless of actual exact values, we could refresh tuples iteratively until the precision constraint is met. Iterative CHOOSE_REFRESH algorithms have been studied in [38] and [62]. It may be interesting to investigate in which contexts an iterative method is preferable to the batch method presented in Chapter 4. Also, we could use an iterative method to give bounded aggregation queries an "online" behavior [49], where the user is presented with a bounded answer that gradually refines to become more precise over time. In this scenario, the goal is to shrink the answer bound as quickly as possible.

Improving Performance

- **Delaying the propagation of insertions and deletions to the central repository.** We plan to investigate ways in which discrepancies in the number

of tuples can be bounded, and the computation of the bounded answer to a query can take into account these bounded discrepancies. Sources will then no longer be forced to send a refresh every time an object is inserted or deleted.

- **Considering ways to amortize refresh costs by *refresh piggybacking* and *pre-refreshing*.** When a (value- or query-initiated) refresh occurs, the source may wish to "piggyback" extra refreshes along with the one requested or required. These extra refreshes would consist of values that are likely to need refreshing in the near future, *e.g.*, if the precise value is very close to the edge of its bound. The amount of refresh piggybacking to perform would depend on the benefit of doing so versus the added overhead. Additionally, it might be beneficial to perform *pre-refreshing*, by sending unnecessary refreshes when system load or communication cost is low that may be useful in future processing.
- **Investigating storage, indexing, and query processing issues over interval approximations.** We plan to study ways in which replicated data objects stored as pairs of bound functions might be compressed. Without compression, each source must keep track of the bound functions for each remotely replicated data object. Compression issues can be addressed without affecting the techniques presented in this dissertation: our CHOOSE_REFRESH algorithms are independent of which bound functions are used or how they are represented, and we have not focused on query processing issues. Efficient indexing schemes for interval data have been proposed in, *e.g.*, [63].

Bibliography

[1] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 34–48, San Francisco, California, May 1987.

[2] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 245–256, San Jose, California, May 1995.

[3] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *Proceedings of the International Conference on Extending Database Technology*, pages 223–240, Avignon, France, March 1996.

[4] D. Agrawal and A. J. Bernstein. A nonblocking quorum consensus protocol for replicated data. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):171–179, 1991.

[5] D. Agrawal and A. El-Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 243–254, Brisbane, Australia, August 1990.

[6] C. Ahlberg and E. Wistrand. IVEE: An information visualization and exploration environment. In *Proceedings of the First Information Visualization Symposium*, Atlanta, Georgia, October 1995.

[7] R. Alonso, D. Barbara, H. Garcia-Molina, and S. Abad. Quasi-copies: Efficient data sharing for information retrieval systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 443–468, Venice, Italy, March 1988.

[8] R. Avnur, J. M. Hellerstein, B. Lo, C. Olston, B. Raman, V. Raman, T. Roth, and K. Wylie. CONTROL: Continuous output and navigation technology with refinement on-line. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 567–569, Seattle, Washington, June 1998.

[9] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2003.

[10] D. Barbara and H. Garcia-Molina. The reliability of vote mechanisms. *IEEE Transactions on Computers*, 36:1197–1208, October 1987.

[11] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 373–387, Vienna, Austria, March 1992.

[12] B. B. Bederson, J. D. Hollan, K. Perlin, J. Meyer, D. Bacon, and G. W. Furnas. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. In *Journal of Visual Languages and Computing*, volume 7:1, pages 3–31, March 1996.

[13] M. Benedikt and L. Libkin. Exact and approximate aggregation in constraint query languages. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 102–113, Philadelphia, Pennsylvania, May 1999.

[14] P. A. Bernstein, B. T. Blaustein, and E. M. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proceedings of*

the Sixth International Conference on Very Large Data Bases, pages 126–136, Montreal, Canada, October 1980.

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998.

[17] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, February 1998.

[18] A. Brodsky and Y. Kornatzky. The LyriC language: Querying constraint objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 35–46, San Jose, California, May 1995.

[19] A. Brodsky, V. E. Segal, J. Chen, and P. A. Exarkhopoulo. The CCUBE constraint object-oriented database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 577–579, Philadelphia, Pennsylvania, June 1999.

[20] A. Bulut and A. K. Singh. SWAT: Hierarchical stream summarization in large networks. In *Proceedings of the 19th International Conference on Data Engineering*, Bangalore, India, March 2003.

[21] A. Cedilnik and P. Rheingans. Procedural annotation of uncertain information. In *Proceedings of the IEEE Visualization Conference*, pages 77–84, Salt Lake City, Utah, October 2000.

[22] M. Cherniack, M. J. Franklin, and S. Zdonik. Expressing user profiles for data recharging. *IEEE Personal Communications: Special Issue on Pervasive Computing*, 8(4):32–38, August 2001.

[23] J. Cho and H. Garcia-Molina. Estimating frequency of change. Technical report, Stanford University Computer Science Department, 2000. http://dbpubs.stanford.edu/pub/2000-4.

[24] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 200–209, Cairo, Egypt, September 2000.

[25] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 117–128, Dallas, Texas, May 2000.

[26] W. S. Cleveland. *The Elements of Graphing Data*. Wadsworth, 1985.

[27] E. Cohen and H. Kaplan. Refreshment policies for web content caches. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Anchorage, Alaska, April 2001.

[28] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[29] T. J. Davis and C. P. Keller. Modelling and visualizing multiple spatial uncertainties. *Computers and Geosciences*, 23(4):397–408, 1997.

[30] M. Derthick, J. A. Kolojejchick, and S. F. Roth. An interactive visualization environment for data exploration. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 2–9, Newport Beach, California, August 1997.

[31] M. Dilman and D. Raz. Efficient reactive monitoring. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, Anchorage, Alaska, April 2001.

[32] L. Do, P. Ram, and P. Drew. The need for distributed asynchronous transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 534–535, Philadelphia, Pennsylvania, June 1999.

[33] T. Dorcey. CU-SeeMe desktop videoconferencing software. *Connexions*, 9(3), March 1995.

[34] J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental crawler. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, China, May 2001.

[35] A. El-Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, June 1989.

[36] D. Estrin, L. Girod, G. Pottie, and M. Srivastava. Instrumenting the world with wireless sensor networks. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Salt Lake City, Utah, May 2001.

[37] T. Feder, R. Motwani, L. O'Callaghan, C. Olston, and R. Panigrahy. Computing shortest paths with uncertainty. In *Proceedings of the Twentiety International Symposium on Theoretical Aspects of Computer Science*, Berlin, Germany, February 2003.

[38] T. Feder, R. Motwani, R. Panigrahy, C. Olston, and J. Widom. Computing the median with uncertainty. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, pages 602–607, Portland, Oregon, May 2000.

[39] Y. Fua, M. O. Ward, and E. A. Rundensteiner. Navigating hierarchies with structure-based brushes. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 58–64, San Francisco, California, October 1999.

[40] A. Gal and J. Eckstein. Managing periodically updated data in relational databases: A stochastic modeling approach. *Journal of the ACM (to appear)*, 2002.

[41] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, 1979.

[42] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, Seattle, Washington, June 1998.

[43] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–159, Pacific Grove, California, December 1979.

[44] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, Massachusetts, 1989.

[45] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[46] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, Washington, D.C., May 1993.

[47] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering*, pages 266–275, Sydney, Austrialia, March 1999.

[48] R. L. Harris. *Information Graphics*. Oxford University Press, 1999.

[49] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. Haas. Interactive data analysis with CONTROL. *IEEE Computer*, August 1999.

[50] J. M. Hellerstein and P. J. Haas. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 171–182, Tucson, Arizona, May 1997.

[51] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.

[52] J. Holliday, R. Steinke, D. Agrawal, and A. El-Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(3), May/June 2003.

[53] A. Householder, A. Manion, L. Pesante, and G. Weaver. Managing the threat of denial-of-service attacks. Technical report, CMU Software Engineering Institute CERT Coordination Center, October 2001. http://www.cert.org/archive/pdf/Managing_DoS.pdf.

[54] Y. Huang, R. Sloan, and O. Wolfson. Divergence caching in client-server architectures. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 131–139, Austin, Texas, September 1994.

[55] D. Huff. *How to Lie with Statistics*. Norton, 1993.

[56] N. Huyn. Maintaining global integrity constraints in distributed databases. *Constraints*, 2(3/4):377–399, 1997.

[57] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, October 1975.

[58] N. Jukic and S. Vrbsky. Aggregates for approximate query processing. In *Proceedings of ACMSE*, pages 109–116, April 1996.

[59] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Network Monitoring (MobiCom 99)*, pages 271–278, Seattle, Washington, August 1999.

[60] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.

[61] J. Kangasharju, Y. G. Kwon, A. Ortega, X. Yang, and K. Ramchandran. Implementation of optimized cache replenishment algorithms in a soft caching system. In *Proceedings of the IEEE Signal Processing Society Workshop on Multimedia Signal Processing*, Los Angeles, California, December 1998. http://sipi.usc.edu/∼ortega/SoftCaching/MMSP98/.

[62] S. Khanna and W. Tan. On computing functions with uncertainty. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 171–182, Santa Barbara, California, May 2001.

[63] H. Kriegel, M. Potke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 407–418, Cairo, Egypt, September 2000.

[64] G. M. Kuper. Aggregation in constraint databases. In *Proceedings of the First Workshop on Principles and Practice of Constraint Programming*, Newport, Rhode Island, April 1993.

[65] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the Web. In *Proceedings of the Twenty-Seventh International Conference on Very Large Data Bases*, pages 391–400, Rome, Italy, September 2001.

[66] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, March 2001.

[67] L. A. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[68] C. T. Lawrence, J. L. Zhou, and A. L. Tits. User's guide for CFSQP version 2.5: A C code for solving (large scale) constrained nonlinear (minimax) optimization problems, generating iterates satisfying all inequality constraints. Technical report TR-94-16r1, Institute for Systems Research, University of Maryland, 1997.

[69] I. Lazaridis and S. Mehrotra. Capturing sensor-generated time series with quality guarantees. In *Proceedings of the 19th International Conference on Data Engineering*, Bangalore, India, March 2003.

[70] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 53–60, Washington, D.C., May 1986.

[71] W. Lipski, Jr. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3):262–296, September 1979.

[72] J. W. S. Liu, K. Lin, W. Shih, and A. C. Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[73] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. DEVise: Integrated querying and visual exploration of large datasets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 301–312, Tucson, Arizona, May 1997.

[74] A. M. MacEachren. Visualizing uncertain information. *Cartographic Perspective*, (13):10–19, 1992.

[75] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, California, February 2002.

[76] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, June 2002.

[77] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985.

[78] M. J. McPhaden. Tropical Atmosphere Ocean Project, Pacific Marine Environmental Laboratory, 2001. http://www.pmel.noaa.gov/tao/.

[79] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10), October 1991.

[80] R. Min, M. Bhardwaj, S. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan. Low-power wireless sensor networks. In *Proceedings of the Fourteenth International Conference on VLSI Design*, Bangalore, India, January 2001.

[81] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 181–194, Cannes, France, September 1997.

[82] S. B. Moon, J. Kurose, and D. Towsley. Packet audio playout delay adjustment: Performance bounds and algorithms. *ACM/Springer Multimedia Systems*, 6:17–28, January 1998.

[83] J. P. Morgenstein. Computer based management information systems embodying answer accuracy as a user parameter. Ph.D. thesis, U.C. Berkeley Computer Science Division, 1980.

[84] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings*

of the First Biennial Conference on Innovative Data Systems Research (CIDR), January 2003.

[85] F. Naumann, U. Leser, and J. Freytag. Quality-driven integration of heterogeneous information systems. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases*, Edinburgh, U.K., September 1999.

[86] A. T. Pang, C. M. Wittenbrink, and S. K. Lodha. Approaches to uncertainty visualization. *The Visual Computer*, pages 370–390, November 1997.

[87] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[88] D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, 1995.

[89] L. L. Peterson and B. S. Davie. *Computer Networks, Second Edition*. Morgan Kaufmann, 2000.

[90] V. Poosala and V. Ganti. Fast approximate query answering using precomputed statistics. In *Proceedings of the IEEE International Conference on Data Engineering*, page 252, Sydney, Australia, March 1999.

[91] G.J. Pottie and W.J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):551–558, May 2000.

[92] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–386, Denver, Colorado, May 1991.

[93] K. Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2):199–226, 1993.

[94] R. L. Read, D. S. Fussell, and A. Silberschatz. A multi-resolution relational data model. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 139–150, Vancouver, Canada, August 1992.

[95] P. Revesz. *Introduction to Constraint Databases*. Springer-Verlag New York, Inc., 2002.

[96] E. A. Rundensteiner and L. Bic. Aggregates in possibilistic databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 287–295, Amsterdam, The Netherlands, August 1989.

[97] Y. Saito and M. Shapiro. Replication: Optimistic approaches. Technical report, Hewlett-Packard Labs, 2002. HPL-2002-33.

[98] G. Salton and C. S. Yang. On the specification of term values in automatic indexing. *Journal of Documentation*, 29:351–372, 1973.

[99] P. Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, February 1998.

[100] S. Shah, A. Bernard, V. Sharma, K. Ramamritham, and P. Shenoy. Maintaining temporal coherency of cooperating dynamic data repositories. In *Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases*, Hong Kong, China, August 2002.

[101] T. Skalicky. Laspack reference manual, 1996. http://www.tu-dresden.de/mwism/skalicky/laspack/laspack.html.

[102] N. Soparkar and A. Silberschatz. Data-value partitioning and virtual messages. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 357–367, Nashville, Tennessee, April 1990.

[103] J. Stewart. *Calculus: Early Transcendentals, Second Edition*. Brooks/Cole, 1991.

[104] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1), January 2002.

[105] B. N. Taylor and C. E. Kuyatt. Guidelines for evaluating and expressing the uncertainty of nist measurement results. Technical report, National Institute of Standards and Technology Note 1297, Gaithersburg, Maryland, 1994.

[106] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[107] E. R. Tufte. *The Visual Display of Quantitative Information, Second Edition.* Graphics Press, 2001.

[108] J. W. Tukey. *Exploratory Data Analysis.* Addison Wesley, 1977.

[109] R. Twiddy, J. Cavallo, and S. M. Shiri. Restorer: A visualization technique for handling missing data. In *Proceedings of the IEEE Visualization Conference*, pages 212–216, Washington, D.C., October 1994.

[110] B. Urgaonkar, A. G. Ninan, M. S. Raunak, P. Shenoy, and K. Ramamritham. Maintaining mutual consistency for cached Web objects. In *Proceedings of the Twenty-First International Conference on Distributed Computing Systems*, Phoenix, Arizona, April 2001.

[111] F. J. M. van der Wel, L. C. van der Gaag, and B. G. H. Gorte. Visual exploration of uncertainty in remote-sensing classification. *Computers and Geosciences*, 24(4):335–343, 1998.

[112] R. van Renesse and K. Birman. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. Technical report, Cornell University, 2001.

[113] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A mechanism for background transfers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, December 2002.

[114] S. Vutukury and J.J. Garcia-Luna-Aceves. A traffic engineering approach based on minimum-delay routing. In *Proceedings of the IEEE International Conference on Computer Communications and Networks*, Las Vegas, Nevada, October 2000.

[115] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, 1991.

[116] C. M. Wittenbrink, E. Saxon, J. J. Furman, A. T. Pang, and S. K. Lodha. Glyphs for visualizing uncertainty in environmental vector fields. *IEEE Transactions on Visualization and Computer Graphics*, pages 266–279, September 1996.

[117] J. L. Wolf, M. S. Squillante, P. S. Yu, J. Sethuraman, and L. Ozsen. Optimal crawling strategies for web search engines. In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 2002.

[118] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 588–596, Orlando, Florida, February 1998.

[119] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[120] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proceedings of the Tenth International Conference on Scientific and Statistical Database Management*, pages 111–122, Capri, Italy, July 1998.

[121] A. Woodruff and C. Olston. Iconification and omission in information exploration. In *SIGCHI '98 Workshop on Innovation and Evaluation in Information Exploration Interfaces*, Los Angeles, CA, April

1998. http://www.fxpal.com/ConferencesWorkshops/CHI98IE/submissions/woodruff.htm.

[122] A. Woodruff, C. Olston, A. Aiken, M. Chu, V. Ercegovac, M. Lin, M. Spalding, and M. Stonebraker. DataSplash: A direct manipulation environment for programming semantic zoom visualizations of tabular data. *Journal of Visual Languages and Computing, Special Issue on Visual Languages for End-user and Domain-specific Programming*, 12(5):551–571, October 2001.

[123] T. Yamashita. Dynamic replica control based on fairly assigned variation of data with weak consistency for loosely coupled distributed systems. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems*, pages 280–289, Vienna, Austria, July 2002.

[124] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[125] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 123–133, Cairo, Egypt, September 2000.