Storage Optimization for Large-Scale Distributed Stream Processing Systems

Kirsten Hildrum[1], Fred Douglis[1], Joel L. Wolf[1], Philip Yu[1], Lisa Fleischer[2], Akshay Katta[3]

[1]IBM T.J. Watson Research Center [2]Dartmouth College [3]Amazon Corporation

Abstract

We consider storage in an extremely large-scale distributed computer system designed for stream processing applications. In such systems, incoming data and intermediate results may need to be stored to enable future analyses. The quantity of such data would dominate even the largest storage system. Thus, a mechanism is needed to keep the most useful data. One recently introduced approach is to employ retention value functions, which effectively assign each data object a value that changes over time [5]. Storage space is then reclaimed automatically by deleting data of lowest current value. In such large systems, there will naturally be multiple file systems available, each with different properties. Choosing the right file system for a given incoming data stream presents a challenge. In this paper we provide a novel and effective scheme for optimizing the placement of data within a distributed storage subsystem employing retention value functions. The goal is to keep the data of highest overall value, while simultaneously balancing the read load to the file system.

Keywords

Storage management, streaming systems, theory, optimization, file assignment problem, load balancing

1 Introduction

Distributed computer systems designed specifically to handle very large-scale data stream processing applications are in their infancy. Several early examples augment relational databases with streaming operations [9, 10, 14, 4]. These systems all process voluminous rates of incoming data streams by performing relational operations (such as database joins) over the incoming data.

The authors have been involved in an ambitious project known as *System S* [7] to build the prototype of a highly scalable distributed computer system to handle complex applications involving enormous quantities of streaming data. We expect the system to consist of tens of thousands of processing nodes, to be able to concurrently support hundreds of thousands of incoming and "derived" streams, and to have a storage subsystem with a capacity of multiple petabytes. Even at these sizes, we expect all key resources of the system to be fully utilized nearly all of the time. In particular, the storage subsystem will be always virtually full, because of the enormous data arrival rate [5].

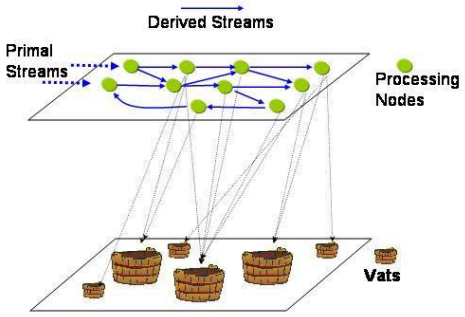


Figure 1. Storage subsystem overview.

In this paper we focus on the distributed storage subsystem, introducing a novel and very effective optimization scheme for such an environment. The storage subsystem of System S is pictured conceptually in Figure 1. Streams are processed by interconnected applications on a distributed set of processing nodes. These processing nodes are interconnected via a network, and also connected, via a separate storage network, to a collection of individual file systems. We refer to these file systems colloquially as *vats*.

Storing streaming data presents a challenge that is qualitatively different from that of conventional systems, because of the huge quantities of primal and derived data that need to be written to disk. The storage subsystem of a conventional computer system is typically configured with sufficient capacity to handle the data. Deletion of data is typically done manually. But in the System S streaming environment massive amounts of data are being written out constantly. As

new data arrives, an equivalent amount of old data must be deleted. Since the deletion operations will happen at great rates, they must also be done automatically.

Easy solutions to the automatic deletion of data might involve *First In, First Out (FIFO)* or *Least Recently Used (LRU)* schemes. In System S we have adopted a more sophisticated approach, originally outlined by Douglis, et al. [5]. This approach treats data differently based on its current importance to the overall system, while clustering data that are anticipated to be deleted together. The data clustering is a performance optimization to avoid the overhead of many file operations on small files and is not germane to this discussion. Here, we motivate why managing objects by value is useful in systems of this scale and emphasize the difficulty in maximizing the system-wide value of data when individual decisions about data placement and retention are made in isolation across multiple vats. The focus of this paper is, in fact, the formulation and solution of precisely this optimization problem via a scheme which considers all of the file systems simultaneously. The need to support time varying values of data is what makes our scenario distinct from those solved by systems such as Minerva [6].

We start with some examples of valuation. As a first example, the transcripts of audio and video streams might be worth storing for longer periods of time than the actual multimedia streams, because they are so much more compact. But newer streams and transcripts are consistently more relevant, and therefore more valuable, than older ones. As another example, derived data from incoming streams may be stored longer than the original streams. For instance, if a process does topic determination of usenet articles or blog posts, the list of topics may be kept much longer than the blog posts or articles themselves.

Many applications would want to use historical data, i.e., data stored before those applications started, and so the system has to store more data than possible. Consider as an example a stream processing system focused on the financial health of companies. It would likely consider both the current stock price and the prices in the past. It might keep quarterly reports, conference calls, news articles, weather information, and video news reports. Now suppose that after the collapse of Enron, a user runs an application to look for warning signs of its collapse. In this case, much of the data would have arrived long before the actual query, and even that particular application would have looked at only a small segment of the stored data. Likewise, a user might use historical data to look for warning signs of the collapse of Internet companies, and then perhaps compare this data to that of current companies.

In these queries, the decision to store and label the data would be made with an anticipation of what information may be generally useful, but without knowing the specific applications which might be involved.

The approach taken in System S is to define for each data object to be written to disk a function describing its projected value over time. This *retention value function* will typically be non-increasing, with a range from 0 to 100. The storage subsystem will then delete the data with the lowest current retention function values as space is needed. (Observe that this design results in a *relative* rather than *absolute* notion of value: the retention function value at a given time does not guarantee the amount of time the data object has left before being deleted.)

The overhead associated with such a deletion scheme is manageable, at least as long as the number of such functions is not too large. The retention value functions are defined at a much coarser level than that of the data objects themselves. We assume that each data object belongs to a *retention class*, and that all data objects in this class have retention values determined by the same retention value function. Thus retention classes are the atomic unit on which retention value functions are defined. Different data objects within a retention class can have varying ages, and therefore have different values at any given time. Occasionally it may be useful to modify a particular retention value function, or to remove certain data objects from a retention class and add them to another, thus changing the retention value functions for those objects. Storage class retention function assignments and data object retention value function modifications are the responsibility of analytics, which work in conjunction with the applications themselves. (One cannot expect the applications themselves to make these decisions without some form of centralized control, because they have a naturally parochial perspective.) The design of such system analytics is in any case orthogonal to the present paper.

Different vats will typically have different properties, and not all retention classes will be suitable for all vats. There are many possible reasons for this. The retention classes might have specific availability, performance, security and/or network locality requirements. The vats themselves may or may not match these. Or the vats may be reserved for specific types of data, such as DB2.

On an individual vat, space is essentially fluid, and deleting existing data frees up space for a comparable amount of new data. As a practical implementation we approximate this concept via a *waterline*. The waterline is defined for a given vat and time: data whose value is below this waterline will be deleted. Data whose value is at or above this waterline will be retained. The waterline rises and falls over time, depending on the amount and nature of the new data that has been added to the vat.

Coping with Distribution The notion of waterlines takes on a much different character when there are multiple vats. Absent some sort of global optimization strategy, it is likely

that the waterlines of the various vats will drift and become quite different over time. But notice that this causes by definition the deletion of higher valued data than would be removed in a scenario with one global vat with a single waterline. What the system should do is approximate, as closely as possible, the case of a single vat, even though not all data is allowed to go to all vats. So it would be ideal if the waterlines of the various vats were as close as possible to identical. But we must also balance the read load to the vats. Some of the incoming data may be predictably "hot" while other incoming data is predictably "cold". The hot data must be placed on the vats that have sufficient capacity to serve the read requests.

Most applications have a choice of vats. Our goal is to ensure, with minimal communication, that the assignment decisions are good for the system as a whole. Periodically, the optimizer will gather information about the data being written and the state of the storage system, and then instruct the applications to revise their vat assignments.

The question of how to do this has the flavor of a traditional file assignment problem (FAP). (An excellent survey is given by Dowdy and Foster [3].) The large majority of FAPs have had the goal of trying to balance load across the storage subsystem. Balancing waterlines instead presents a different challenge. These FAPs have generally made decisions about initial data placement and periodic data movement. Proper initial placement is relatively more critical in a system such as System S. That is because data movement is less useful from a cost/benefit analysis perspective in such a system: first of all, data may only be read a few times before being deleted, so the overhead of movement is high relative to its expected utility. Secondly, movement of data is simply more expensive in a distributed storage system. So we are forced to make very careful initial placement decisions, and treat data movement as expensive (and consequently limited), or even as prohibited. Fortunately, as will be seen, our scheme behaves nearly as well when data movement is not allowed at all.

Formally, the objective of our optimization scheme is to minimize the total value of all data deleted, subject to reasonable and practical constraints. Minimizing the total value of the deleted data is equivalent to maximizing the total value of the data retained. This goal is achieved by making optimal decisions about where to write newly created data, and also how to move data around within the storage subsystem, provided such movement is within the limits allowed.

We next focus on the constraints of the problem. The first constraint corresponds to a key rationale for the vats themselves: different vats will typically have different properties, and not all retention classes will be suitable for all vats. For example, vats may have different availability properties (e.g., RAID level), performance properties (e.g., access latency), security properties, different physical locations, qualitative properties (e.g., some vats might be reserved for DB2 data). Each retention class may have specific requirements with respect to these properties, and thus be allowed only on a subset of the vats. The optimization scheme must allocate newly created data only to a vat which is acceptable (that is, it meets *all* of the requirements). The optimization scheme may move existing data only from one acceptable vat to another acceptable vat. Second, the scheme must obey various constraints describing (at either a local or a global level) the maximum amount of such movement. Finally, it must ensure than no vat gets too many requests for reads. (This constraint is analogous to the load balancing objective of traditional FAPs.)

We believe that the optimization scheme described in this paper is the first of its kind. It is being integrated into System S, and it requires minimal centralized control and direction. It is *epoch*-based, which means that it wakes up at the beginning of each epoch, gathers data, computes and implements a new solution, and then sleeps until the end of the epoch. The exact length of an epoch is not crucial, and one could imagine trying to optimize this length as well. We do, however, expect to use epochs of roughly a half hour. In practice the time to do the optimization is on the order of a minute.

The rest of this paper is organized as follows. Section 2 describes the mathematical formulation of our optimization problem, as well as the proposed solution. It also outlines several alternative implementations. Section 3 describes some additional approaches of a simpler nature, and compares all of the approaches using simulation experiments. These experiments show our scheme to be both effective and practical. In Section 4, we draw conclusions and outline future work.

2 Problem Formulation

Consider a finite collection of M retention classes denoted by $r \in \{1, ..., M\}$. These retention classes can correspond to existing data on disk, to new data being written to disk, or to both. We also have a finite collection of N vats denoted by $v \in \{1, ..., N\}$. For ease of notation we also employ a vat 0 corresponding to new data, not yet assigned to an "actual" vat.

For simplicity we will first introduce the problem without load balancing. Then we will describe a formulation that addresses read capacities.

We also define the following constants:

- $Z_{r,v}$ is the amount (in bytes) of retention class r data in vat v. In particular, $Z_{r,0}$ is the amount of new data in retention class r.
- C_v is the capacity (in bytes) of vat v.

- $A_{r,v}$ is 1 if retention class r is allowed in vat v, and 0 otherwise. The M x $(N+1)$ matrix A is called the *acceptability* matrix.

- $c_{v,v'}$ is the (per byte) cost of moving data from vat v to vat v', $v, v' \in \{1, ..., N\}$; naturally, $c_{v,v} = 0$.

- $k_{v,v'}$ is the maximum amount of data (in bytes) that can be moved in one epoch from vat v to vat v', $v, v' \in \{1, ..., N\}$.

- K is the maximum amount of data (in bytes) that can be moved between all vats in the system in one epoch.

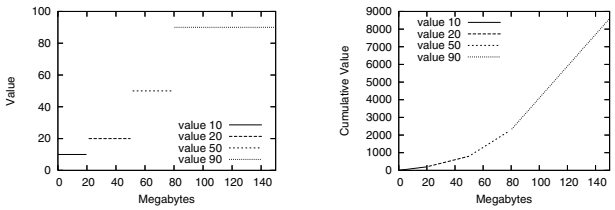
For simplicity, we make the assumption in what follows that all vats are *full*. This will be nearly true, as indicated, almost all of the time. Very simple changes are required to deal with vats that are not completely full.

In order to formulate the optimization problem, we define a function $V_{r,v}$ for each $r \in \{1, ..., M\}$ and $v \in \{0, ..., N\}$. Specifically, $V_{r,v}(x)$ is the cumulative amount of value lost when x bytes are deleted from retention class r in vat v. (When $v = 0$ it will represent new data that is deleted immediately, and thus never stored.) These $V_{r,v}(x)$ depend on the age of the retention class r data in vat v and the shape of the data's retention curve. The independent (x-axis) variable of $V_{r,v}$ will represent the *amount* of data (in bytes) from retention class r which will be deleted from vat v to accommodate new or existing data entering the vat. The dependent (y-axis) variable of $V_{r,v}$ will represent the *cumulative value* of the data deleted. Because deletion removes data of smallest value first, we start by ordering the data in terms of increasing value per byte for each retention class r and vat v. This gives rise to a (preliminary) function $W_{r,v}$ defined as the value $W_{r,v}(w)$ of the (last) object removed if a total of w bytes are deleted. Assuming the values are discrete (integers, say, from 0 to 100), $W_{r,v}$ is a step function with one step for each different value of data in the vat. (See Figure 2a.) Then the function $V_{r,v}$ is the integral of this function between 0 and w: $V_{r,v}(w) = \int_0^w W_{r,v}$. Because of the nature of $W_{r,v}$, the function $V_{r,v}$ is an increasing and piecewise linear convex function of w as shown in Figure 2b.

We will formulate our optimization problem as a linear program (LP). The intuition for this LP comes from a flow graph, which is not shown due to space restrictions. In certain special cases the problem will actually be solvable via network flow techniques. See for example, Bertsimas and Tsitsiklis [2] and Ahuja, et al. [1], respectively.

We now define two types of decision variables:

- $y_{r,v,v'}$ is the amount of data from retention class $r \in \{1, ..., M\}$ that will be moved from vat $v \in \{0, ..., N\}$ to vat $v' \in \{1, ..., N\}$. Note that $y_{r,v,v}$ is the amount of data in retention class r that remains in vat v.



(a) Value lost. (b) Cumulative value lost.

Figure 2. The functions $W_{r,v}$ and $V_{r,v}$

- $w_{r,v}$ is the amount of new or existing data from retention class $r \in \{1, ..., M\}$ that will be deleted from vat $v \in \{0, ..., N\}$.

We are now ready to describe two alternative optimization problems. The first of these does not deal with load balancing. The second does.

Optimization formulation without load balancing The optimization problem formulation, which we call **LP1**, is as follows:

$$\min \left(\sum_{r,v} V_{r,v}(w_{r,v}) + \sum_{v} \sum_{v'} c_{v,v'} \sum_{r} y_{r,v,v'} \right) \quad (1)$$

subject to

$$w_{r,v} + \sum_{v' : A_{r,v'}=1} y_{r,v,v'} = Z_{r,v} \qquad \forall (r, v) \quad (2)$$

$$\sum_{r : A_{r,v}=A_{r,v'}=1} \sum_{v'} y_{r,v',v} = C_v \qquad \forall v \quad (3)$$

$$\sum_{r : A_{r,v}=1} y_{r,v,v'} \leq k_{v,v'} \quad \forall v \neq 0, v' \neq v \quad (4)$$

$$\sum_{r, v \neq 0, v'} y_{r,v,v'} \leq K \quad (5)$$

$$w_{r,v}, y_{r,v,v'} \geq 0 \qquad \forall r, v, v' \quad (6)$$

The objective function 1 includes summands for the value of deleted data and for the cost of moving data from vat to vat. By scaling the cost coefficient $c_{v,v'}$ we can easily vary the importance of one relative to the other. Equations 2 and 3 are "flow conservation" constraints. Inequalities 4 are the local movement constraints, and inequality 5 is the global movement constraint. Constraints 4 and 5 turn the optimization problem into an LP rather than a network flow problem. Inequality 6 is the non-negativity constraint.

There are a couple of relevant special cases, both of which are network flow problems. First, if the movement is unrestricted and has no cost, all data in the system is treated as new data, reducing the number of variables. Second, if *no* movement can occur, the problem is a matter of placement and not data migration.

Optimization formulation with load balancing Finally, we enhance the previous formulation to address load balancing. Load balancing is a common goal in traditional FAPs. In our environment it must be balanced against our prime goal of balancing the waterlines.

- For $r \in \{1, ..., M\}$, d_r is the expected read access rate for data in retention class r.
- For $v \in \{1, ..., N\}$, D_v is the maximum read access rate threshold for vat v.

Beyond the threshold D_v, the performance decreases sharply. Effectively, this is the knee of the performance curve. There are many alternative techniques, both empirical and theoretical, for determining D_v. A discussion of these issues appears in [13]. The choice is orthogonal to the current paper, as are the forecasting techniques required to obtain the values d_r.

Recall the formulation of the previous subsection. It would be ideal if, after the assignment of new data and the movement of existing data, we could meet this load balancing goal perfectly. For example, we might wish to have $\sum_{r,v'} y_{r,v',v} d_r / D_v$ identical for all vats v. This will not generally happen by accident; it will not even necessarily be possible. We will revise the formulation to attempt to come closer to this load balancing goal. We explored two different approaches and present one here.

The approach involves modifying the LP formulation. We will need one more variable, γ, which represents the read load of the vat as a fraction of its read capacity.

We now revise the objective function of the previous formulation as follows.

$$\min \alpha \left(\sum_{r,v} V_{r,v}(w_{r,v}) + \sum_{v} \sum_{v'} c_{v,v'} \sum_{r} y_{r,v,v'} \right) + (1-\alpha)\gamma \tag{7}$$

Here α is a constant between 0 and 1, and the objective function is now a weighted average of the old objective function and γ.

We also add new access density constraints.

$$\sum_{r,v'} y_{r,v',v} d_r \leq \gamma D_v \quad \forall v \tag{8}$$

$$\gamma \geq 0 \tag{9}$$

Inequality 8 bounds the factor by which we will *miss* our load balancing goals by γ, and inequality 9 is the non-negativity constraint. But the objective function in 7 attempts to minimize γ. This mathematical setup means that γ is the relative load of the vat with the highest relative load. We call this new optimization problem **LP1-alpha**. The objective function can be weighted towards either the original or new objective. If $\alpha = 1$, then the load balancing goal becomes irrelevant. If $\alpha = 0$ the problem has the flavor of a traditional FAP.

3 Simulation Experiments and Results

In this section we describe the results of our simulation experiments. First, we investigate the impact of employing estimates of incoming data rates (instead of the exact values) as we would naturally have to do in practice. We discover that there is virtually no loss in performance by using reasonable estimates. Second, we consider the impact of data movement. If movement is completely free, data can be migrated from one vat to another at will. This yields an upper bound on the benefits of such data migration. Conversely, if migration has infinite cost, movement is not allowed at all.This is a lower bound on the benefits of migration. We observe that the differences in performance (in terms of waterline differentials and skewed read loads) between the no movement and free movement cases are not significant in these simulations. The conclusion is that allowing data movement is for all practical purposes unnecessary. Finally, we compare our LP-based algorithms with some natural heuristics for this storage management problem.

Setup We simulate the effect of the various algorithms on a storage system with 200 retention classes and 20 vats. We choose this number because it represents the likely scale of a large stream processing system. Recall that the 20 vats each represent individual *file systems*, not individual disks. Furthermore, there is a natural limit to the number of retention classes that can be used. Assigning a particular piece of data to even one of fifty retention classes would be difficult; two hundred is an overestimate.

In each step, approximately 1.75% of the total storage capacity in the system arrives, displacing an equal percentage of the data currently in the system. (This choice means that the system achieves stability after a reasonable number of iterations.) The amount of data arriving in each retention class is normally distributed around a mean fixed for each run. The standard deviation is set equal to the mean.

To model the distribution of data arrival means and number of acceptable vats, we use Zipf [15] and Zipf-like [12] distributions. These distributions are commonly used to model file sizes and access patterns, and are generally thought to represent the real world well. Specifically, we use a Zipf-like distribution with parameter θ_{accept}, so that the number of acceptable vats for the ith retention class is proportional to $i^{1-\theta_{accept}}$. If $\theta_{accept} = 1$ the number of acceptable vats is identical for the various retention classes. It $\theta_{accept} = 0$ the distribution is pure Zipf. To vary the skew in the data arrival rates amongst the retention classes, we

use a parameter θ_{arrive}, so that the amount of data in the ith highest retention class is proportional to $i^{1-\theta_{arrive}}$.

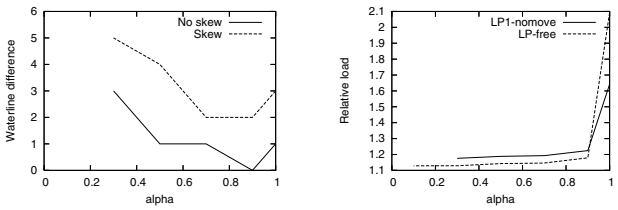
For the graphs shown in this paper, both θ_{arrive} and θ_{accept} are 0.5, so both distributions are moderately skewed. We also produce graphs with $\theta_{arrive} = \theta_{accept} = 1$, but space limitations precluded presenting both sets of graphs. Thus, we only present the more realistic skew case here. The uniform graphs are qualitatively similar, but the differences between schemes are usually a bit smaller.

We chose retention value functions with roughly three phases. In the first phase, the value is high but declining slowly; in the second, the value is declining quickly; and in the third, the value is low and declining slowly. The value of an object that is t time units old is given by $v(t) = c * (-\arctan((t-p)/a) + \pi/2)$ where p, a, and c are parameters fixed for a given retention class. The parameter p describes how long the value remains steady, a describes how sharply it declines in value, and c is a constant used to set the initial value. We use two values of a and three values of p, which might reasonably correspond to one, two and seven days. (We also ran experiments with other choices of retention functions, and the results were qualitatively similar.) The function $V_{r,v}(w_{r,v})$ can be computed from $v(t)$ knowing the amount of bytes at age t each retention class r in vat v.

The graphs that follow were generated by a simulator written in java, with linear programs solutions via CLP [11], part of the COIN [8] project. In the simulation runs, each algorithm is given the same incoming data at every iteration. However, because different algorithms make different deletion decisions, the set of data in the vats at a given time may be different among the various algorithms. In all the simulation experiments, the setup starts with an "empty" storage system. During the start up phase the vats are filling, and the waterline differences become very high because some vats fill before others. The graphs in this paper typically start when the vats have filled, at approximately iteration 200.

To measure effectiveness at balancing waterlines, we graph the difference between the waterline of the vat with the smallest waterline and the waterline of the vat with the largest waterline. The bigger the difference, the larger the quantity of data that should be kept that is in fact deleted because of the random vat choice.

To ensure that the heavily accessed streams are well-distributed, we use a metric we call the relative load. We assume each vat has a maximum acceptable number of accesses, D_v, corresponding to the "knee" of the response time curve. The relative load is the maximum over all vats of the total accesses to that vat divided by D_v. The relative load is the same as γ of **LP1-alpha**. Thus, if the relative load of a configuration is 0.8, then the most heavily loaded vat is using 80% of its capacity.



(a) Waterline Difference, both (b) Relative Load, skew

Figure 3. Affect of trade off factor α

Balancing the objectives In this subsection, we compare ways of incorporating balancing between the waterline and access density of objectives, and conclude that there is a clear winner. Recall that the access density is incorporated into our algorithms we described in Section 2 by adding constraints 8.

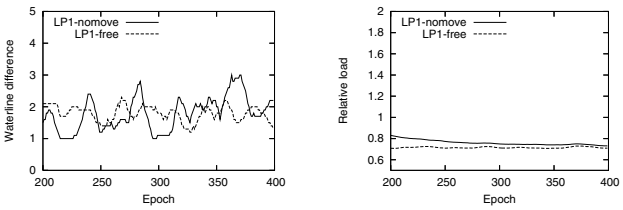
Figure 3 shows both the waterline difference and the relative load for different values of α. Notice that is a trade off between waterline difference and the maximum relative load. However, the relative load drops substantially as α goes from 1.0 to 0.9, with only a small increase in the waterline difference. This suggests that **LP1-alpha** with $\alpha = 0.9$ is the best choice. An α below 0.9 does not substantially improve the relative load objective, and it hurts the waterline objective significantly.

Note that as α decreases, emphasizing the relative load objective, in the no-movement case, the relative load can actually go up. Our simulator follows the movement and placement suggestions from the LP, but always deletes the data of lowest value. For low α values, the LP answer may suggest deleting data which is above the lowest value. As a result, the simulator may not implement the solution of the LP. This is intentional as it would not be sensible in a real system to delete data because it is getting too many accesses. (One might throttle accesses to the data instead, for example.) From these graphs, we conclude that using the **LP1-alpha** with $\alpha = 0.9$ is the best in this scenario.

Sensitivity of LP1 In this section, we show that **LP1** is robust to estimated data arrival rates and also is relatively insensitive to the amount of data movement. Thus allowing data movement is not essential to achieve good algorithmic performance, an important statement in practical terms.

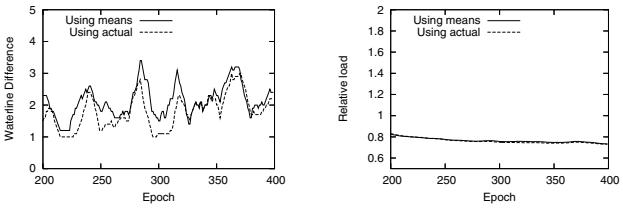
As noted, the goal is to keep the waterlines as equal as possible and simultaneously balance the access rates. To measure the closeness to this goal, we plot the maximum waterline difference and the relative load, averaged over the last ten epochs. Figures 4 compares **LP1** with unlimited, no-cost movement (**LP1-free**) to an **LP1** algorithm with no movement allowed (**LP1-nomove**)

The graphs show that the waterline difference for the



(a) Waterline difference, skew (b) Relative load, skew

Figure 4. Free movement and no movement



(a) Waterline difference, skew case. (b) Relative load, skew case.

Figure 5. Exact vs estimated arrivals

LP1-nomove case is no more than one away than **LP1-free** with $\alpha = 0.9$. Because of the acceptability constraints, the optimal waterline difference is not always zero or one. Mathematically speaking, the optimal waterline difference in most problem instances will be at least one due to integrality issues, more if the acceptability constraints are sufficiently difficult. Solutions which have a waterline difference of one or less can be regarded as nearly or exactly perfect. They may be so even if the difference is greater.

Incoming data estimates. The optimization algorithm of Section 2 takes as input the amount of incoming data in each retention class. This will invariably not be known exactly in advance. In Figure 5, we show that using approximations produces results nearly as good as using exact values (these graphs allow no movement of placed data, though the free movement case is similar). Comparing the "mean" line on the rightmost graphs shows that allowing movement does not reduce the waterline difference when the data arrivals are only approximate.

In both cases, the amount of data arriving is generated using a normal distribution. For convenience, let $Z^*_{r,0}$ be the amount of new data in retention class r. (Recall that $Z_{r,0}$ the amount of new data in retention class r, and is given as input to **LP1**. Also, $y_{r,0,v}$ is a variable in **LP1** that represents the amount of this sent to vat v). For those curves labeled "actual," **LP1** is given this number precisely, in that $Z_{r,0}$ (the input to the linear program) is $Z^*_{r,0}$. However, knowing $Z^*_{r,0}$ precisely is unrealistic. For those curves labeled "means," $Z_{r,0}$ is the expected value of $Z^*_{r,0}$. The LP returns values $y_{r,0,v}$ that describe how to place the data. However, the system actually has to place $Z^*_{r,0}$ rather than $Z_{r,0}$ bytes of data. So we interpret $y_{r,0,v}/Z_{r,0}$ as the proportion of the $Z^*_{r,0}$ sent to vat v.

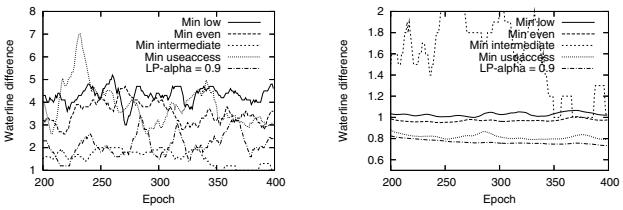
The differences between the "actual" and the "means" cases are noticeable, but fairly small. This holds true in both the skew and (unshown) uniform cases, when movement is free, and regardless of constraints on movement. Thus, we conclude that **LP1** performs well even with only estimates of incoming data rates.

Comparison to other techniques In this section, we compare the linear programming approach described above to natural heuristic approaches. The first algorithm, denoted **even**, divides the incoming data of a particular retention class in proportion to the storage capacity of the vats that accept it. We will show that this algorithm performs very badly with regard to the waterline metric, resulting in some vats with waterlines near one hundred, and others with waterlines near zero. The second algorithm, denoted **min**, sends a data chunk to the vat with the waterline of lowest numerical value. A tie-breaking rule used when there is more than one such vat affects performance. Ties are fairly frequent, since the waterline is measured discretely, using a granularity of one percent.[1] Thus the tie-breaking rule actually matters. We consider three rules. The first, **min-low**, always break ties in favor of the lowest numbered vat. The second, **min-even** breaks ties evenly among all tied vats. The **min-access** algorithm uses the access densities to proportionally break ties. Finally, the **min-intermediate** algorithm uses the even tiebreaking rule, but gets fresh waterline information after each placement. (The others always use the waterline information from the beginning of the epoch for the duration of the epoch.) The intermediate second model may be more realistic, corresponding to a system in which writers reevaluate their decisions at uncorrelated intervals, while the other model is more pessimistic.

The **even** algorithm has no natural way to incorporate these access rates. For the **min** algorithm, we will use the access rates as part of the tie-breaking rule, as follows. We send data with high d_r values ("active" data) to under loaded vats, and less active data to overloaded vats.

Figure 6 compares the performance of the **even** algorithm, the **min** algorithm, and the **LP1** algorithm using the waterline difference and relative load. Since all versions of the **LP1** perform similarly, we use the (most restrictive) no movement version with α set to 0.9. In terms of waterline, the **even** algorithm performs quite poorly compared to the

[1] We believe this granularity is a reasonable representation of a large distributed system. Delays in sending and gathering information will render a more exact statement of the waterline meaningless; thus, we believe that it is natural to use a model that incorporates some rounding.



(a) Waterline difference, skew case.

(b) Relative load, skew case.

Figure 6. even(off scale), min, and LP1

rest. In fact, it is actually off the scale. (The **even** algorithm performs somewhat better for different sets of retention decay functions, but it is always far above the other options.) In the relative load metric, it is the second worst performer, behind the unadorned LP that does not take relative load into account.

The **LP1** algorithm has the smallest waterline difference. The performance of the **min** algorithm varies, depending first on the tie-breaking rule used, and second on how current the waterline information is. The lowest line for the **min** algorithm assumes that the waterline information is updated after each placement. In terms of waterline, this algorithm performs comparably to the **LP1** algorithm. On the access rate objective, it is a significantly worse performer. While the **LP1** algorithm is using about 70% of the capacity of the most heavily loaded vat, the **min** algorithm is using about 80%. If the data were uniformly more active, that difference could be the difference between 95% and 105%, which would have a big impact on system performance. For the relative load metric, the **LP1** algorithm is the winner.

4 Conclusions and Future Work

We have shown how to balance two objectives in a distributed storage system. Our two objectives are (1) to balance the access rates to the vats, ensuring that no one vat is overloaded, and (2) to ensure that the highest-value data is kept. We give several linear programming based solutions and compare these to natural heuristic approaches. Of these, we find that **LP1-alpha** with $\alpha = 0.9$ balances the two objectives best of the solutions considered.

There is much work still to be done, both in terms of our storage optimizer and System S. Though the storage optimizer works well on synthetic data, it needs to be integrated into the rest of the system and tested in the prototype environment. The final system will probably include several additional constraints of a practical nature: for example, we need to bound the number of changes allowed to the storage class/vat decisions in each epoch. This is not hard, but the effectiveness of the scheme must be tested given these additional constraints. We know the optimization scheme is fast enough and will scale well, but we need to develop a variety of infrastructure to make it work properly. In particular, we need to incorporate forecasting and modeling techniques to supply accurate input data. We need to ensure that the solutions are implementable in real time. Finally, we need to decide on an optimal length for a storage optimizer epoch.

References

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice Hall, 1993.

[2] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[3] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computer Surveys*, 14(2):287–313, 1982.

[4] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR 2005 - Second Biennial Conference on Innovative Data Systems Research*, 2005.

[5] F. Douglis et al. Position: Short object lifetimes require a delete-optimized storage system. In *Proceedings of 11th ACM SIGOPS European Workshop*, 2004.

[6] G. A. Alvarez et al. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[7] L. Amini et al. Adaptive control of extreme-scale stream processing systems. In *Proceedings of ICDCS 2006*, 2006.

[8] R. Lougee-Heimer et al. The COIN-OR initiative: Open-source software accelerates operations research progress. *ORMS Today*, 28(5):20–22, 2001.

[9] S. Chandrasekaran et al. TelegraphCQ: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, 2003.

[10] S. Zdonik et al. The Aurora and Medusa projects. *IEEE Data Engineering Bulletin*, 26(1), 2003.

[11] J. Forrest. CLP- COIN-OR linear program solver. http://www.coin-or.org/Clp/index.html, 2006.

[12] D. E. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, 1973.

[13] S. Lavenberg, editor. *Computer Performance Modeling Handbook*. Academic Press, 1983.

[14] The STREAM Group. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1), 2003.

[15] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.