Time-varying Management of Data Storage

Ranjita Bhagwan, Fred Douglis, Kirsten Hildrum, Jeffrey O. Kephart, William E. Walsh
IBM T. J. Watson Research Center

Abstract

The efficiency of large-scale applications is strongly dependent on good data management techniques. In this paper, we claim that the ability to specify data requirements in a time-varying *manner facilitates data management and improves application efficiency. This is because requirements such as availability, bandwidth and latency can vary significantly with time. Consequently, the storage system can dynamically change the allocation of resources to data objects. We describe how an application may specify these dynamic requirements using utility functions, and outline a strategy towards achieving an optimal allocation of resources to data objects.*

1 Introduction

Applications depend heavily on good data management to function properly. However, managing data efficiently has become a very complex problem. Many factors contribute to this, including voluminous data storage, varied application requirements that can change over time, and disparate storage characteristics.

First, the amount of data stored by applications is growing at a phenomenal rate. It is estimated that many enterprises are witnessing growth in storage needs at the rate of 70 to 120% per year [9]. This explosion is not limited just to the corporate world. New scientific experimentation techniques generate extremely large datasets whose sizes are estimated to go up to exabytes within the next decade [5]. Emerging data mining applications [8] are estimated to generate large amounts of data that will keep any contemporary storage system in a state of constant overflow.

As a consequence, the common assumption that storage is abundant is applicable only to some domains. Although disk sizes are rapidly increasing, storage devices in some environments will eventually reach capacity. The current approach to this problem is to adopt some combination of the following three strategies: add more storage, move data from a primary device to a secondary or archival device, or delete data. But which of these actions should we pursue? Which data do we move or delete? Where and when do we move data? Currently, administrators set these relevant parameters manually, though such a policy can lead to undesired system behavior. For example, if a data item that an application considers important gets moved to archival storage, it may adversely affect the application's performance.

Second, applications have different requirements for the different kinds of data, and these requirements may *change with time*. For example, newly generated web server logs may be critical to an application such as a real-time intrusion detection system, hence these data will have significant availability and performance requirements. It may therefore be useful to generate multiple replicas of these logs on storage devices with low latency and high throughput. But with time, their utility to the application, hence their availability and performance requirements, diminish. Hence older web logs would be suitable candidates to delete or move to archival store as storage systems reach capacity.

Third, there are a multitude of storage devices with widely different operational parameters. Systems can support low latency and high bandwidth, but at a high cost compared to other options. Archival media offer low costs but much poorer performance in the worst case. Replication can be used to provide both high availability and improved performance, again at a cost.

We are working on a large-scale distributed stream processing system in which these questions of data management are critical. One assumption in the design of this system is that its capacity is chronically insufficient, and prior work has focused primarily on automatically selecting data to retain based on predetermined specifications of how the relative value of the data varies over time [8]. This is particularly appropriate for data mining applications in which the product of the available data and the set of potential mining algorithms dwarf any conceivable set of storage resources. Here we take initial steps to extend the earlier retention framework [8] to consider other time-varying criteria. We make the following observations:

Applications should be allowed to specify their requirements to the storage system in a time-varying manner. Particularly, information that an application may provide is related to desired retrieval performance (latency limits and bandwidth required to retrieve stored data), availability requirements, and estimated access patterns to the data objects. The system should take these requirements and map them into storage-level parameters such as which storage devices to place the data on and how many replicas to make. In addition, applications should inform the storage system of how these parameters change with time, so that the system can make informed decisions on moving or replicating the data if required in the future.

When a storage device is full, the system should take into account these requirements to determine data movement strategies to free space and redistribute load. Moving objects that have minimal requirements improves overall system efficiency. Thus, the system can strive towards a data placement strategy that gets as close to application requirements as possible. At the same time, the system will still need to consider the outright deletion of low-value data when their value declines over time.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 explains a scenario that we will use in the rest of the paper to elucidate our proposed approach to managing storage load. In Section 4 we describe what an application may specify as part of data requirements using time-variant functions of the requirements and *utilities* associated with them. Section 5 gives an overview of a proposed optimization strategy for storage management using these specifications.

2 Related work

Several efforts such as Hippodrome [1], WiND [2], SLEDRunner [6] and Chameleon [16] have concentrated on adaptive techniques in storage management to meet performance requirements. PASIS [18] addresses the problem of providing security and availability in distributed storage systems. These systems assess, to a certain extent, how object these requirements change by explicitly monitoring storage systems. While this approach is effective in some cases, in others, the application's knowledge of how data requirements are going to change should be used. Consequently, providing the storage system with the time-varying requirements improves and simplifies data management. Moreover, our work concentrates on the fact that in some scenarios, storage systems reaching capacity is a common occurrence. Hence data management has to explicitly deal with this problem. Other research efforts [11] have evaluated the use of tertiary storage for enhancing performance of specific applications such as video-on-demand. We believe that the incorporation of time with these strategies can significantly improve data management.

Information Lifecycle Management (ILM) [14] and Policy-based Lifecycle Management [17] are efforts to identify that data items typically have lifetimes that decay with time. Recent work [8] extends this idea to annotate every object with a "retention" value that decays over time. Our effort further extends the notion of retention by addressing properties that applications would be directly interested in such as availability and performance.

3 Example scenario

In this section, we provide two examples to elucidate why application-specific, time-varying specification of data requirements is useful. The first example emphasizes data availability while the second concentrates on required performance. We then give an example of a storage layer that would manage data given these requirements.

3.1 Scenario 1: Insider trading detection

Consider a system that attempts to correlate multiple input sources to identify possible cases of insider trading. These might include:

Stock quotes. An abrupt change in value would suggest the opportunity to exploit any advance knowledge.

News items. News about a company frequently results in changes to its stock price. Significant changes that result from news that some people had advance knowledge of are of great interest in detecting insider trading, but unforeseeable news is not. News may take the form of video, audio, web pages, etc.

Transactions. To identify insider trading, one must record the buy and sell transactions.

Each of these types of data has different storage requirements. Stock quotes are transient: when one sees a sequence of quotes for a stock at approximately the same amount, only the most recent one need be retained. Other data are less predictable. For instance, if a "foreseeable" news event results in a sudden change in stock value, not only are the news and the changing stock quotes important, so are all transactions in that stock over the period when the news was known to insiders. The older transactions are, the less likely they will be tied to insider trading, so their importance declines over time, as does any requirement to store those transactions in a redundant fashion. In this case, the emphasis of our approach for that application is on retention and availability.

3.2 Scenario 2: Sporting event web server

Consider a sporting event web service that provides information such as news, results, player profiles, and video streams. Such a web service creates several data items including HTML documents, media streams and web logs. Again, we are faced with varying storage requirements, some of which pertain to retention. For example, system administrators will most likely use web logs within a few

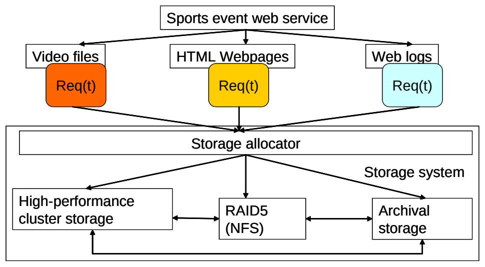

Figure 1. Sporting event web server

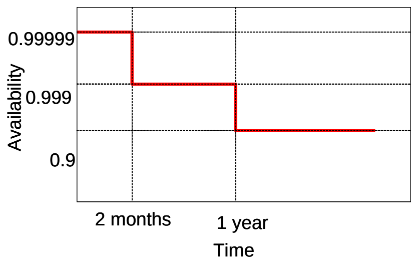

Figure 2. A time-varying availability specification

hours after they are created or not at all. Hence, the importance of preserving these web logs diminishes with time.

On the other hand, while video streams and HTML documents both require more persistent storage than logs, they differ in terms of performance requirements. Multimedia streams need to be delivered at a certain rate to ensure reasonable quality, whereas users can tolerate higher latencies for HTML documents. During an event, the number of accesses to documents related to the event will increase accordingly. Hence some documents such as live multimedia streams will need more resources at certain times.

3.3 Data management

In Figure 1, we show how Scenario 2 interacts with an example storage system. The storage resources include an NFS-style file system with RAID, a high-performance cluster file system, and a larger tape archive. The storage allocator uses the application specifications to map data objects to storage units.

Consider the case that the high-performance cluster reaches capacity and the application generates new data to be inserted into this high-performance storage unit. The storage system therefore decides to free up some space in the cluster file system by moving objects that have become less important (older transactions in the first scenario and older web logs in the second) to the archive. Since it is wasteful to move data that is soon to be accessed to the archive (reading objects off the archive involves large overheads [11]), the data management strategy should evaluate these trade-offs while choosing an allocation policy.

Our goal is to generalize this model by identifying different ways an application can specify its requirements and by laying out storage system mechanisms that can ensure high performance and availability.

4 Specification

In this section, we elaborate on the different requirements that an application can specify to the storage system. The goal of the storage system is to adhere as closely as possible to the application's requirements while performing efficient data management. The following are four axes along which application requirements vary with time.

Availability: As in the examples above, required data availability is mainly a function of how an application wants to use the data. Consequently, an application can specify availability per data type as a function of time. When a data object is created and is inserted into the storage system, it is accompanied by an availability requirement as shown in Figure 2. In this figure, availability is measured as the probability that a request for the object is satisfied. With the passage of time, required availability of stock quotes, for example, can diminish, and the system can accordingly free up resources. Allowing applications to specify availability requirements in this way allows the system to treat objects differently over time. Without such a specification, the system is forced to keep at least 0.99999 availability forever because this was needed early in the object's lifetime.

Performance: Performance requirements of data can vary based on data type. In the example of Section 3.2, multimedia streams will require higher performance than textual data. However, performance will also depend on the nature of the application. In the first example, if an added functionality of the application were to detect illicit trades in real-time so that they could be stopped proactively, it would impose certain performance requirements on data access. It is more probable that the analysis of newly created data will lead to such discoveries. Hence, performance requirements of data can also be expressed as a function of time, with the units of performance being retrieval time or latency and bandwidth. In the case of multimedia files, bandwidth is more important, while the important metric in the case of real-time anomaly detection is the latency.

Access patterns Accesses to data can be correlated with time. While in the first example, it is reasonable to believe that newer data is accessed more often, in the second example, the actual time or date of an event affects the popularity of an object. When creating data for an event that occurs on say the 10th of July, the application can specify a higher popularity for the event on the 10th of July.

Application importance Multiple applications may use the same storage system and certain applications may be more "important" than others. Using techniques such as service-level objectives [3], the system may assign priorities to each application. It must respect this priority and the availability, performance and access specifications when placing data.

4.1 Utility of meeting requirements

While the framework outlined thus far supports automated data retention decisions by allowing applications to specify simple time-varying storage requirements for data it is useful to augment the framework further to allow applications to characterize the relative desirability of certain storage attributes. Accordingly, we allow applications to express how important it is to them that the storage system meet their (possibly multi-dimensional) requirements by specifying a *utility function* for each storage object. Figure 3 illustrates simple time-varying utility as a function of allotted bandwidth for two objects, O_1 and O_2. In April, O_1 needs at least 350 Kbps in order to achieve significant utility, while O_2 is relatively satisfied with bandwidth above 100 Kbps. In May, O_1's utility U_1 shifts to a lower asymptotic value and a lower required bandwidth. We elucidate how the storage allocator employs these utility functions to make data management decisions in Section 5.1.

4.2 Specifying utility functions

In principle, unique utility functions could be defined for each data object. However, we advocate grouping data objects into a finite set of classes, such that each object in a class shares the same utility function. This approach reduces the cognitive burden on administrators because it permits them to specify a much smaller number of utility functions. Furthermore, the grouping can reduce the computational burden on the allocation optimizer, as explained in Section 5.2. Possible class discriminators may include file format (mpeg files require higher performance than html files as described in Section 3.2), or data semantics (stock quotes may be less important than news articles, as shown in Section 3.1). Specifying the time-dependency of utility functions introduces some added burden for administrators, but can be reasonably managed by predefining a small set of functions for each class. Effectively, each class is partitioned dynamically into a small set of sub-classes: objects within a certain sub-class are approximately of the same age and use the same utility function. This is similar to policy-based lifecycle management [17], however, we believe that utility-function specification should be at a higher level and not rigid, so that the storage allocator can take into account the condition that storage devices may be operating at capacity or near-capacity. The allocator may therefore make decisions that will not give the application exactly what it desires, but tries to get as close to the requirements as possible. Section 5.1 gives a simple example of how these different utility functions can be used to capture the time-dependency of data requirements.

Even with this simplification, eliciting utility functions from the administrator and assigning them to classes remains a key problem. Quite generally, preference elicitation is a challenging and open area of research [10]. We envision it as an iterative process, whereby a user interface presents a series of questions to the administrator, asking her to express preference trade-offs. An example question could be "Do you prefer $[50\text{Kbps}, 400\text{Kbps}]$ or $[350\text{Kbps}, 100\text{Kbps}]$", where $[\text{bw}_1, \text{bw}_2]$ is a possible allocation of bandwidth between class 1 and and class 2. We can use recent techniques [13] to obtain an accurate estimate of the utility function with minimal elicitation. Since we cannot query the administrator indefinitely, the resulting utility function will likely not reflect the administrator's preferences perfectly. Small discrepancies between the administrator's true preferences and the elicited utility function should in most cases lead to allocations whose *value* is close to that of the allocation that the administrator would have chosen, although the *allocation* itself may be significantly different. If the system permits the administrator to provide occasional feedback on allocation decisions, then over time the system can incrementally learn utility functions that better reflect the administrator's preferences.

5 Optimization

In this section, we describe how the storage system can use optimization to place data objects and allocate resources in accordance with application requirements. The optimization is complicated by the time-varying application requirements and the continual flow of new data into the system.

The main component of the system is the *storage allocator*, which performs the optimization that maps requirements associated with data objects, as represented by utility functions, to storage units based upon their properties. Higher availability requires storage on more available devices and more replicas. Higher performance requirements can translate to more replicas or placement on storage devices that have high bandwidth and low latency.

The allocator needs to know the state and properties of the storage units in the system, such as available bandwidth, latency and availability parameters such as MTBF. For a new object, it determines what would be a suitable placement of the object, i.e., how many replicas to make, and which storage units will store these devices. Because some storage units may be nearly full, or there may be insufficient available bandwidth, the optimization must consider moving or deleting pre-existing objects.

In the next subsection we provide a simple example of using optimization to place data objects with time-varying

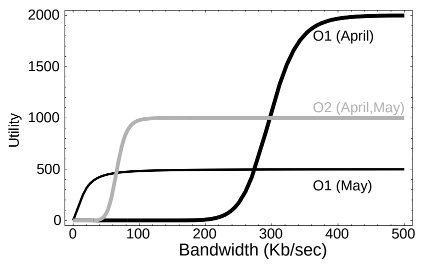

Figure 3. Time-varying utility as a function of bandwidth for objects O_1 and O_2.

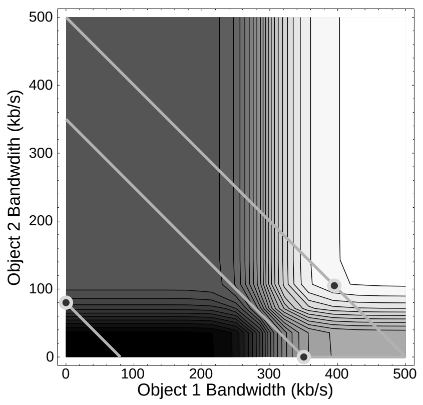

Figure 4. Feasible utility regions. Lighter colors indicate higher utility. Curved dark lines indicate iso-utility curves. Gray, straight lines indicate different bandwidth constraints of storage devices in the system.

utility functions. Following that, we discuss some practical issues that arise in optimizing large-scale storage systems.

5.1 Simple example

To illustrate our approach, consider two objects O_1 and O_2, each of which can be placed on storage units S_H or S_L. Storage unit S_L provides a low bandwidth of 80 Kbps, while S_H provides a higher bandwidth—either 350 Kbps or 500 Kbps. Suppose further that the bandwidth requirements of objects O_1 and O_2 are characterized by time-varying utility functions as given in Figure 3. Specifically, in April, O_1 has much higher bandwidth needs than O_2 (roughly 300-400 Kbps, as compared with roughly 60-100 Kbps), and its utility is twice that of O_2. However, a month later, O_1's bandwidth needs are reduced below those of O_2, and its maximum utility is half that of O_2. O_2's needs do not change.

The storage allocator may use a simple optimization algorithm to determine the objects' optimal placement, as well as the optimal apportionment of bandwidth if O_1 and O_2 are placed on the same storage unit. (Here we assume that a QoS control mechanism as described in Chambliss et al. [6] is in effect, so that one can enforce bandwidth limitations on O_1 and O_2 individually.) The algorithm considers all feasible placements and bandwidth allocations, and identifies one that maximizes the total utility $U_1 + U_2$.

Figure 4 is a topographic map of the function $U_1 + U_2$ in the month of April. Higher utility is represented by lighter-color regions, while the dark lines represent curves along which the utility has a constant value. The gray, straight lines represent different bandwidth constraints; the one closest to the origin with intercepts at 80 Kbps on the horizontal and vertical axes represents the bandwidth constraint for S_L, while the two further from the origin represent bandwidth constraints for S_H (either 350 Kbps or 500 Kbps). For a given bandwidth constraint, the intersection of the constraint line with the highest-valued iso-utility curve gives the optimum solution; these are indicated by points.

Now suppose that it is April, and that S_H provides a bandwidth of 350 Kbps. There are four placement possibilities for the optimizer to consider. First, consider placing both O_1 and O_2 on S_L. The optimal bandwidth allocation would be 0 Kbps for O_1 and 80 Kbps for O_2, as indicated by the point in Fig. 4. This would yield a total utility $U_1 + U_2 = 850$. Note that, even though O_2 has a lower bandwidth requirement and a lower utility, it is favored because O_1 would derive essentially no benefit from the full bandwidth of 80 Kbps. At this bandwidth, O_2 is not fully satisfied, but it receives a reasonable fraction of its maximum utility. Second, consider placing both O_1 and O_2 on S_H. Then, as shown in Fig. 4, the optimal allocation would be 350 Kbps for O_1 and 0 Kbps for O_2, yielding a total utility of 1813. Third, consider placing O_1 on S_H and O_2 on S_L. Then O_1 would receive 350 Kbps and O_2 would receive 80 Kbps, resulting in a total utility of $1813 + 850 = 2663$. Fourth, and finally, consider placing O_1 on S_L and O_2 on S_H. This would yield a total utility of 1000. Thus the optimal placement and allocation are attained for the third case: O_1 on S_H and O_2 on S_L, resulting in total utility 2663.

If S_H's capacity were increased from 350 Kbps to 500 Kbps, the optimizer would derive a different placement: both O_1 and O_2 would reside on S_H, with 395 Kbps allocated to O_1 and the remaining 105 Kbps allocated to O_2. The overall total utility would be $1962+986 = 2948$, somewhat better than the $1999 + 850 = 2849$ that would be attained for the placement that is optimal in the previous case.

Now suppose that a month passes, taking the scenario where S_H's capacity is 350 Kbps. The shift in U_1 (Fig. 3) would result in a dramatic change in the topography of Fig. 4. The optimization algorithm would determine that O_2 should be moved from S_L to S_H, with 204 Kbps allocated to O_1 and 146 Kbps allocated to O_2 for a total utility of $495 + 999 = 1494$. One could even contemplate adding more storage objects like O_2 to S_H. Considering

S_H only, the overall maximum utility of 3846 occurs when there are four O_2's on S_H, each with bandwidth allocation of 83 Kbps, with the single O_1 receiving the remaining 19 Kbps. Additional copies of O_2 would result in overcrowding and overall degradation in the total utility. Thus, in addition to determining placement, optimization based on utility functions can be used to make automated decisions about whether to retain objects in storage, or admit more of them.

5.2 Practical issues

In a large system with many data objects, optimization is a challenge. While finding the global optimal solution may be infeasible, we believe that a good solution can be found in a reasonable amount of time. As noted in Section 4.2, objects will belong to a storage class, each with its own utility function. The system can then, instead of allocating space between objects, allocate space among the storage classes, significantly reducing the size of the problem. Typically, utility functions will be nonlinear, but the inter-class optimizations can be solved very efficiently if the utility functions are convex [4]. Utility functions will often not be convex, as in Figure 3, in which case it is necessary to use methods that do not do not assume convexity, such as the Nelder-Mead algorithm [12], trust-region algorithm [7] or minimax-regret optimization [13]. The intra-class optimizations can be solved independently, and we believe we can exploit similarities of objects within a class to do this efficiently, albeit approximately. To cope with the dynamic nature of the system, continual reoptimization is necessary. For the sake of efficiency, such reoptimization can be performed incrementally, with occasional recomputation from scratch.

There are at least two different ways of incorporating the time-varying nature of the utility functions into the optimization problem. This is complicated by the fact that moving objects from one storage device might maximize value, but comes with a cost. One solution is to simply re-optimize at every given point in time, trying to achieve the solution with the highest value minus cost. An alternative approach is to take a more long-term point of view and allocate taking into account the future data movements, and attempting to maximize total long-term value, perhaps with a Markov Decision Process [15] approach. However, this requires knowing (or assuming) more about the future data.

6 Summary

We proposed a new direction for data management in large storage systems in which applications can specify how their requirements vary with time. We outline how applications may specify these requirements and describe an optimization strategy for the data placement. We believe that this approach will allow the storage system to make more informed decisions towards efficient data management.

Acknowledgments

We thank Paul Dantzig, Murthy Devarakonda, David Grabiner, Nagui Halim, Joe McCloskey, John Palmer, David Pease, Elizabeth S. Richards, David Tao, Bill Tetzlaff, Sandeep Uttamchandani and Honesty Young for their helpful feedback.

References

[1] E. Anderson et al. Hippodrome: Running circles around storage administration. In *Proc. of FAST*, 2002.

[2] A. Arpaci-Dusseau et al. Manageable storage via adaptation in WiND. In *Proc. of (CCGrid)*, 2001.

[3] L. L. Ashton et al. Two decades of policy-based storage management for the IBM mainframe computer. *IBM Systems Journal*, 2003.

[4] S. Boyd and L. Vandenberghe. *Convect Optimization*. Cambridge University Press, 1994.

[5] J. J. Bunn and H. B. Newman. Data-intensive grids for high-energy physics. In *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2002.

[6] D. D. Chambliss et al. Performance virtualization for large-scale storage systems. In *Proc. of SRDS*, 2003.

[7] A. R. Conn, N. Gould, D. Orban, and P. L. Toint. A primal-dual trust-region algorithm for non-convex nonlinear programming. *Mathematical Programming*, 87(2):215–249, 2000.

[8] F. Douglis et al. Position: Short Object Lifetimes Require a Delete-Optimized Storage System. In *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

[9] Business data is growing by 70 to 120 percent a year: Are you prepared? http://www.embarcadero.com/support/data_explosion.pdf.

[10] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, 1993.

[11] M. Kienzle et al. Using tertiary storage in video-on-demand servers. In *Proc. of IEEE CompCon*, 1995.

[12] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[13] R. Patrascu et al. New approaches to optimization and utility elicitation in autonomic computing. Technical report, University of Toronto and IBM, 2005.

[14] M. Peterson. Information lifecycle management: A vision for the future. http://www.snia.org/tech_activities/dmf/SRC-Profile_ILM_Vision_3-29-04.pdf.

[15] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 1994.

[16] S. Uttamchandani et al. CHAMELEON: A self-evolving, fully adaptive resource arbitrator for storage systems. In *Proc. of USENIX*, 2005.

[17] A. Verma et al. An architecture for lifecycle management in very large file systems. In *Proc. of IEEE/NASA MSST*, 2005.

[18] J. J. Wylie et al. Selecting the Right Data Distribution Scheme for a Survivable Storage System. Technical Report CMU-CS-01-120, Carnegie Mellon University, May 2001.