Satisfying Database Service Level Agreements while Minimizing Cost through Storage QoS

Frederick R. Reiss

Computer Science Division
University of California
Berkeley, CA 94720
Email: phred@cs.berkeley.edu

Tapas Kanungo

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
Email: kanungo@almaden.ibm.com

Abstract

The emerging paradigms of service oriented architectures and utility-based computing have the potential to greatly reduce the cost of data management. Data management service providers negotiate with their customers to agree on a service level agreement (SLA) that guarantees performance and reliability. However, these providers have the freedom to cut costs by taking advantage of economies of scale across multiple customers.

In this paper, we examine the problem of choosing a QoS level for each table or index in a service provider's backend databases so as to minimize the dollar *cost of provisioning storage while satisfying application-level SLAs. This problem is difficult because changes in the access cost of different portions of the database can cause the database to alter its access patterns. We develop an algorithm that optimizes the choice of query execution plans and storage layout* simultaneously *to meet an SLA at minimum cost.*

In our experiments we use a part of the TPC-H benchmark as the workload and several models of the incremental cost of placing a volume at a high quality of service. Our results show that significant cost savings are possible through selective use of high storage QoS levels.

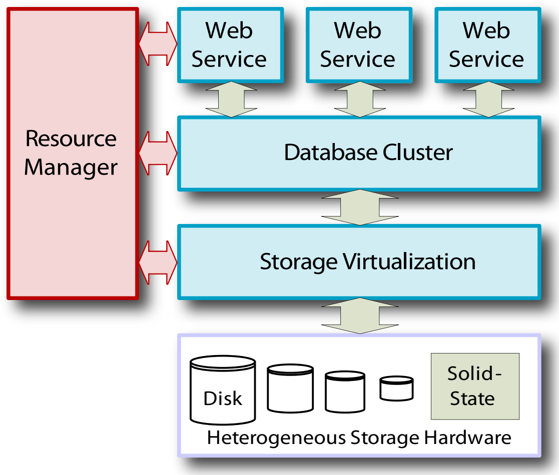

Figure 1: Architecture of an outsourced web service provider. The provider maintains web services on behalf of several clients. The services share a local cluster of relational databases servers, and a storage virtualization layer allocates on-disk storage to these databases. A global resource management framework allocates CPU and database resources among services and storage resources among database instances.

1 Introduction

A recent trend in enterprise computing is the use of service-oriented architectures for managing business data. By using service description languages like WSDL [5] and standard architectures like OSGA [23], businesses can provide easy and transparent access to their data, both for performing internal processes and coordinating with partners, customers, and suppliers.

This standardization has spawned a new industry in web service outsourcing. Because different services use the same platforms and interfaces, companies like IBM [17] and HP [15] can manage web services on behalf of many clients at a single centralized facility. These providers can realize significant economies of scale by sharing hardware and software resources among many clients (See Figure 1).

Currently, outsourcing web services is a manual process. The client negotiates workflow descriptions, service level agreements (SLAs) and pricing with the provider. Then the provider implements and hosts the web service. Research prototypes in the Grid community are moving towards automating the service creation process with technologies like GRAM [14] and SRM [28]. In either case, service outsourcing clients solicit bids from different providers.

To provide competitive bids, providers need a reliable way to minimize the cost of provisioning their resources to meet service level agreements. For data-intensive business applications, meeting a service level agreement typically requires performing a mix of relational database queries within a specified amount of time. The performance of this query mix in turn depends on the speed with which the database can fetch the relevant records from secondary storage.

Service providers need to allocate their storage resources to ensure that database queries execute sufficiently quickly. This provisioning problem is complicated by the fact that there are often several ways to obtain the answer to a relational query. Each of these *query execution plans* can stress different on-disk data structures. For example, a database might have the option of fetching records directly from a relation or indirectly using an index. The optimal storage provisioning depends on the query execution plan, and the optimal query execution plan depends on how storage is provisioned.

In this paper, we examine the storage provisioning problem for services that store their data on relational databases. We show how to optimize the choice of query execution plans and storage QoS levels *simultaneously* to meet an application-level SLA at minimum cost.

2 Background

To allow storage-intensive applications like databases to share a single device, researchers have developed storage systems that provide Quality of Service (QoS) guarantees [3, 30, 19, 20]. With a QoS-enabled storage manager, a service provider can guarantee that each database will always receive a certain amount of bandwidth and latency from a centralized storage device.

QoS does not come for free, however. Internally, storage management software meets QoS guarantees through techniques such as overprovisioning time-shared resources like network bandwidth; moving high-priority data to faster disks or solid-state cache; replicating data across several devices for higher read bandwidth; and leaving portions of a disk drive empty.

There can be a significant dollar cost to increasing the QoS level of a volume; depending on whether disk or network bandwidth is the limiting factor in the storage system's performance, it may be necessary to add additional disks or network interfaces.

Because setting QoS levels high is expensive, it is desirable to do so only for those areas of storage that are important to application performance. For applications with static access patterns, one can determine the performance-critical portions of storage by examining traces of application execution. Modern databases, however, have *query optimizers* that adapt storage usage patterns to the relative performance of different volumes, choosing query execution plans that favor volumes with higher bandwidth and lower latency.

This adaptivity creates additional opportunities for cost savings. Different parts of a database can be of widely varying sizes, and placing the smaller on-disk data structures at high QoS levels can be considerably less expensive than doing the same for larger data structures. If a database provider configures these smaller data structures for higher performance, the query optimizer will react by altering its choice of query plan to favor the smaller data structures. In this way, the database can meet the requirements of its service level agreement with a less expensive combination of service QoS levels.

In this paper, we examine the problem of choosing the optimal set of QoS guarantees for the storage underlying a relational database. We pose the problem as a cost-minimization problem in which the independent variables are the storage QoS levels for each volume in the database. The objective is to meet response time requirements for a given query workload while minimizing the amount of additional hardware that must be added to the storage system. We model the marginal cost of QoS for storage systems that are network-limited, disk-limited, or a combination of the two. We also take into account the behavior of the database query optimizer choosing different query execution plans depending on the relative speed of accessing different portions of the database.

In the next section we compare our work to related literature. In Section 4, we provide our theoretical framework for our analysis. Next, in Section 5.3 we show that the SLA satisfaction problem can be posed as a mathematical optimization problem. To demonstrate our methdology, we conducted experiments using the TPC-H workload and the experimental design described in Section 6. Finally, the economic impact of our automatic provisioning algorithm is discussed in Section 7.

3 Related Work

The problem of allocating resources within a storage system to meet a priori quality of service requirements has received much attention in the storage community in recent years. Alvarez *et al.* [1] describe a system that uses constraint-based heuristic optimization to design storage layouts that meet complex performance requirements. Anderson *et al.* [2] present a similar system that uses a constraint solver to select configuration parameters for RAID logical volumes based on performance specifications. Our work differs from this previous work in that we study database query response time and account for different storage parameters resulting in different query plans.

The field of query optimization [27, 29] is as old as relational databases themselves. The techniques in this paper use a traditional query optimizer as a subroutine and therefore take advantage of this earlier body of work. Some tools for physical database design [11, 26, 24] use a similar architecture to ours to solve the problem of choosing indexes.

The problem of resource allocation to maximize a utility function is well-studied in the field of economics; an optimization algorithm for resource allocation under conditions similar to those investigated in this paper is described in [9].

4 Theoretical Framework

In this section, we describe a theoretical model of a relational database system performing queries using data structures that reside inside centralized storage. This formalizing is similar to that in [25, 16, 12, 11]. Later in the paper we use this model to set up the database layout problem as a cost-based optimization problem.

The rest of the section is organized as follows. First, we model the workload characteristics and the storage system cost functions. Next, we model query plans as vectors whose components are resource usage amounts. The notion of linear cost now becomes a matter of taking vector products of resource usage vectors and storage cost model vectors. This linear cost model is then used in the query optimization process.

4.1 Storage and Workload Models

We model the database workload as a set of ordered pairs (q, w_q), $q = 1, \ldots, Q$. where q is an index to the qth relational database query, and w_q is the query's associated weight. These weights can be used to either model the frequency with which queries appear in the workload, or the relative importance of queries to customers. The service provider can compute the query workload for a given application from a trace of the application's calls to the database.

The queries in this workload run over a set of out-of-core data structures $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ such as data tables or B-Tree indexes. Data structure D_i takes up σ_i units of data on disk.

The database provider can choose a different *quality of service* for the storage partition that holds each data structure. We assume that there are L different service levels, which we index by l, $l = 1, \ldots, L$.

Associated with each service level l is a *transaction time* parameter $T_t(l)$, which indicates the average time that elapses between the database requesting a transfer of data and the transfer commencing. In this context, the term *transaction* refers to a single request for one or more blocks of data from the storage system. The transaction time parameter models queueing time, network latency, seek time, and rotational latency.

Each service level l also has an associated data transfer rate which can be used to calculated the *time per datum* $T_d(l)$. This quantity models the time required to transfer a unit of data to or from the storage utility once the transfer operation has begun.

We assume that the service levels are totally ordered; that is, $T_t(1) < T_t(2) < \ldots < T_t(L)$, and $T_d(1) < T_d(2) < \ldots < T_d(L)$.

We model the cost of provisioning different qualities of service by assigning a *storage cost* C_l^S and *network cost* C_l^N to each service level l. Storage cost is measured in units of money per gigabyte stored, and network cost is measured in units of money per gigabyte transferred.

Each mapping from data structures to service levels defines a function $\alpha : D \rightarrow \{1, 2, \ldots, L\}$. That is, data structure D_i resides on storage with quality of service level $\alpha(D_i)$.

For each such mapping α, we define a *resource time cost vector*

$$T_\alpha = \left(T_t(\alpha(D_1)), \ldots, T_t(\alpha(D_n)), T_d(\alpha(D_1)), \ldots, T_d(\alpha(D_n))\right)^\top \quad (1)$$

Intuitively, the first n elements of the resource time cost vector model the time required to begin transferring data to or from data structures D_1 through D_n, and the next n elements model time required to transfer a unit of data to or from those data structures.

4.2 Query Plans

For each query q, the database system can use any of a number of query plans $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$. Each query plan P_i has an associated *resource usage vector*

$$U_i = \left(U_t^1(i), U_t^2(i), \ldots, U_t^n(i), U_d^1(i), U_d^2(i), \ldots, U_d^n(i)\right)^\top \quad (2)$$

where n is the number of out-of-core data structures. Element $U_d^j(i)$ of the resource usage vector indicates how many units of data the query plan transfers to and from data structure j. Likewise, element $U_t^j(i)$ of the vector indicates how many times the query plan will initiate such data transfers.

4.3 Linear Cost Model

We assume that query execution time adheres to a linear cost model similar to that used by Selinger et al. [27] and most commercial query optimizers and that this execution is I/O-bound. That is, the time to execute a query plan p

(the *time cost* of p) under service level mapping α is equal to

$$U_p^\top \cdot T_\alpha \tag{3}$$

where U_p and T_α are vectors as defined in Sections 4.1 and 4.2 and $\cdot$ is the dot product operator. Intuitively, this equation converts the query plan's counts of storage transactions and data units transferred into units of time by multiplying these counts by the constants in T_α. We ignore the CPU time that a query plan requires.

4.4 Query Optimizer Model

We assume that the database system uses a relational *query optimizer* to choose a query plan for each query that minimizes the time cost (See Equation 3.) of the query. For a given query q and service level mapping α, the query optimizer searches the space of all possible query plans, estimates their resource usage vectors, and picks the plan with the lowest estimated time cost under α. We denote this *estimated optimal plan* by p_o, so the resource usage vector (See Section 4.2.) of the optimal plan is U_{p_o}.

Of course, a query optimizer can make inaccurate estimates of resource usages if it has inaccurate estimates of the selectivities of relational operators. For the purposes of this paper, we assume that the optimizer has a method of obtaining accurate selectivity estimates. Intuitively, the storage provisioning analysis we propose is a long-running offline process, so the optimizer has time to obtain accurate estimates of the operator selectivities by running different query plans on a sample of the database.

A relational query optimizer typically considers a very large set of potential query plans when optimizing a query. However, only a subset of these plans can ever become the optimal plan as a result of changes in storage performance. We call this set of plans the *candidate optimal* plans.

More formally, a query plan p with resource usage vector U_p is *candidate optimal* if there exists a mapping α and a resource time cost vector T_α such that, for any query plan p' with resource usage vector $U_{p'}$, $U_{p'} \cdot T_\alpha \leq U_p \cdot T_\alpha$.

5 Cost-Based Optimization

In the previous section we introduced the notation and representation for storage systems, workloads, and linear cost models. In this section we will use the representation to pose the database layout problem as a quadratic optimization problem. First we define the weighted time and weighted dollar costs and then we setup the constrained optimization problem based on these cost models.

5.1 Weighted Time Cost

Using the workload model from Section 4.1 and the query optimizer model from Section 4.4, we define the *weighted time cost* of a query workload (q, w_q) under a service level mapping α and *query plan mapping* $\beta : q \rightarrow \mathcal{P}$ as:

$$\begin{aligned}\tau(\alpha,\beta) &= \sum_{q=1}^{Q} w_q \times (\text{Time to execute query } q) \\ &= \sum_{q=1}^{Q}\sum_{d=1}^{D} w_q \times (T_t(\alpha(D_d)) \times U_t^d(\beta(q)) \\ &\quad + T_d(\alpha(D_d)) \times U_d^d(\beta(q)))\end{aligned}$$

where β is a mapping from queries to query plans.

Intuitively, the weighted time cost is proportional to the expected running time of the average query in the workload.

5.2 Weighted Dollar Cost

We model the incremental cost of placing data at a QoS level as having two components:

- A *storage price* component that models the disk resources required for each gigabyte of additional data stored.
- A *network price* component that models the additional network resources required per gigabyte transferred.

Depending on the configuration of the storage system, one of these two components will tend to dominate. For example, if there is ample disk bandwidth but network bandwidth is limited, the network price will be the dominant price. The important distinction between these two types of cost is that storage price varies with the amount of data stored on disk, while network price varies with the amount of data *transferred*.

Each service level l has its own storage and network prices, which we denote by C_l^S and C_l^N, respectively.

Using these cost constants, we define the *fixed dollar cost* of a mapping α as

$$\sum_{d=1}^{D} \sigma_d \times C_{\alpha(D_d)}^S. \tag{4}$$

Recall that data structure D_i takes up σ_i units of data on disk. Intuitively, the fixed dollar cost models the cost of keeping data on storage with a particular QoS guarantee, regardless of how much that data is accessed.

Conversely, the *variable dollar cost* of service level mapping α and query plan mapping β is

$$\sum_{q=1}^{Q}\sum_{d=1}^{D} w_q \times U_d^d(\beta(q)) \times C_{\alpha(D_d)}^N. \tag{5}$$

5.3 The Optimization Problem

The database administrator needs to choose a mapping from data structures to service levels that minimizes dollar cost while meeting an average response time constraint. That is, given the current workload (q, w_q) and the maximum response time T_0, he or she needs to choose a mapping functions α and β so as to minimize the total cost $g(\alpha, \beta)$ subject to the constraint that $\tau(\alpha, \beta) < T_0$.

Let $\alpha_{dl} \in \{0, 1\}$ be a binary variable that indicates the assignment of a data structure (a table or an index) to a storage service class. If there are D dats structures and L service levels, then $d = 1, \ldots, D$ and $l = 1, \ldots, L$. Furthermore, if α_{dl} is non-zero it implies that data structure d is assigned to service level l. Each row in the $0-1$ matrix α_{dl} contains only one non-zero entry.

Similary, let the number of queries be Q and the number of possible plans for query q be P_q. Let $\beta_{qp} \in \{0, 1\}$, where $q = 1, \ldots, Q$ and $p = 1, \ldots, P_q$ be a binary variable that indicates which plan is being used for a particular query. For example, if $\beta_{79} = 1$, it implies that plan number 9 is being used for query 7. Also notice that since only one plan can be used for a query, each row of the β_{qp} should have only one non-zero entry. The optimization problem can now be stated as:

$$\begin{aligned}
\text{minimize } g(\alpha,\beta) \quad &= \quad \sum_{d=1}^{D}\sum_{l=1}^{L}\alpha_{dl}C_l^S\sigma_d + \\
& \quad\quad \sum_{d=1}^{D}\sum_{l=1}^{L}\sum_{q=1}^{Q}\sum_{p=1}^{P_q}\alpha_{dl}\beta_{qp}w_qU_t^d(p)C_l^N \\
\text{subject to} \quad & \\
T_0 \quad &\geq \quad \sum_{q=1}^{Q}\sum_{d=1}^{D}\sum_{l=1}^{L}\sum_{p=1}^{P_q}\alpha_{dl}\beta_{qp}w_qT_t(l)U_t^d(p) \\
& \quad\quad +\sum_{q=1}^{Q}\sum_{d=1}^{D}\sum_{l=1}^{L}\sum_{p=1}^{P_q}q\alpha_{dl}\beta_{qp}w_qT_d(l)U_d^d(p) \\
\sum_{l=1}^{L}\alpha_{dl} \quad &= \quad 1 \text{ for all } 1 \leq d \leq D \\
\sum_{p=1}^{P_q}\beta_{qp} \quad &= \quad 1 \text{ for all } 1 \leq q \leq Q \\
U(p)\cdot T(l) \quad &\leq \quad U(p')\cdot T(l) \text{ for all } p \neq p' \text{ iff } \beta_{qp} = 1 \\
\alpha_{dl} \quad &\in \quad \{0,1\} \\
\beta_{qp} \quad &\in \quad \{0,1\}
\end{aligned}$$

The above problem is a integer quadratic programming problem with binary variables. This problem is well studied in the literature [21] and there are numerous software packages available to solve such problems [22, 4].

While the above formulation is correct, we were not able to find public domain optimzation packages that could solve the above integer quadratic programming problem. In Appendix A we use a trick that converts the problem into a *linear* binary programming problem by variable transformation. There are public domain optimization packages available for such problems.

6 Experiment Design

Using the theoretical framework from Section 4, we designed an experiment that measures the potential cost savings from the selective use of high storage QoS. We then ran this experiment with the DB2 query optimizer [6, 13] and the queries from a subset of the TPC-H benchmark [8].

Our experiment proceeded in three stages:

1. Find all candidate optimal plans (See Section 4.4.) and their resource usage vectors (See Section 4.2.).

2. Remove plans that will not occur under the current service levels.

3. Use the remaining plans and service level cost to compute the least expensive mappings from data structures to service levels.

6.1 Experimental Setup

In choosing our experimental setup, we attempted to model a real-world database. Towards this end, we used the query optimizer from a leading commercial database, IBM DB2, and a database schema and queries from a well-known database benchmark, TPC-H.

6.1.1 Database Design and Statistics

We used the database schema and optimizer statistics from an actual published run of the TPC-H benchmark [7] for the computations in our experiments. In particular, we obtained statistics from the 100 GB database used in this benchmark run and loaded these statistics into the system catalogs of IBM DB2 Version 8.1.

We configured the database system to divide the storage resources of the schema in as fine-grained a manner as possible. The DB2 query optimizer "saw" each table and its indexes as residing on a different simulated storage devices. With seven TPC-H tables and a temporary space, our experimental setup had a total of 15 out-of-core data structures.

6.1.2 Queries

We ran our analysis on six of the queries from TPC-H. In particular, we used queries 1, 4, 6, 12, 17, and 19. We chose these queries to maximize coverage of the TPC-H benchmark while using a set that let our linear programming solver complete in approximately five minutes for each data point in our plots. The number of candidate optimal plans for these queries ranged from 2 to 28. We weighted all the queries evenly.

6.1.3 Service Levels

We used three storage QoS levels in our experiments. The highest QoS level had 10 times the performance of the next highest level and 100 times the performance of the lowest level.

Since different storage systems may have different bottleneck components, we considered several possible models for the marginal cost of placing an out-of-core data structure at a given service level:

- The *storage only* cost structure models a storage system in which disk bandwidth is the major constraint. In this model, the network cost of each service level is zero. The price per megabyte of managed storage is proportional to the bandwidth of each service level.

- The *network only* cost structure models a storage system whose performance is constrained by the amount of network bandwidth available. In this model, the storage cost of each QoS level is zero, while the cost of transferring data to and from managed storage is proportional to bandwidth.

- Under the *network and storage* cost structure, network and storage costs are both proportional to bandwidth and are normalized to be approximately equal. This cost structure models a storage system in which both disk and network bandwidth are heavily utilized.

We chose the the queueing time and time per datum parameters (See Section 4.1.) of each service level such that they were always related by the same constant multiple. This constraint allowed us to group the queueing and data transfer elements of our resource usage vectors into a single dimension, halving the dimensionality of these vectors.

6.2 Stage 1

The first stage of our experiment computed the set of candidate optimal plans for all of the queries, as well as computing their resource usage vectors. We conducted this stage using the software described in [25]. Briefly, this software works in three passes:

1. Sample the resource time cost space (See Section 4.1.) through iterative subdivision similar to a high-dimensional quad-tree [10].

2. Augment the set of time cost vectors from Pass 1 until there are enough vectors to calculate resource usage vectors for all known plans by Gaussian elimination.

3. Verify that Pass 2 found all candidate optimal plans by computing and checking the vertices of all the region in which each plan is optimal. If new candidate optimal plans are found, go back to Pass 2.

The software from [25] is compatible with any query optimizer that allows users to set the cost of accessing different storage devices and to retrieve the estimated total cost of the current optimal plan. However, this software takes several hours to run. For a real-world implementation of the ideas in this paper, the implementor should use parametric query optimization [16] [18] to compute the set of candidate optimal query plans more quickly.

6.3 Stage 2

The first stage of our experiment computed the set of plans that could become optimal under any set of storage time costs. With a finite set of storage service levels, only certain storage time costs are possible. The second stage of our experiment removed candidate optimal plans that could not become optimal under the discretized set of storage time costs. This stage worked by solving a set of linear constraints for each candidate optimal plan.

Recall that a given query plan P_i is optimal if, for all other plans P_j for the same query, the time cost of P_i is less than or equal to the time cost of P_j. Therefore, P_i can become optimal under a set of service levels L if there exists a mapping α from data structures to service levels such that

$$U_i \cdot T_\alpha \leq U_j \cdot T_\alpha \text{for all } j \neq i, \tag{6}$$

where U_i, U_j, and T_α are vectors as defined in Sections 4.1 and 4.2.

As in Section 5.3, we converted these constraints into a Boolean linear programming problem by adding an indicator variable $\alpha_{dl} \in \{0, 1\}$ for each data structure d and each service level l, where $\alpha_{dl} = 1$ if data structure d is assigned to service level l. Stage 2 of our experiment solved this rewritten problem using the OPBDP solver [4].

6.4 Stage 3

The final stage of our experiment computed the optimal mappings from data structures to storage service levels under different cost structures, using the candidate optimal plan sets generated in Stage 2. Stage 3 worked by solving the linear program described in Section 5.3 using the OPBDP solver [4].

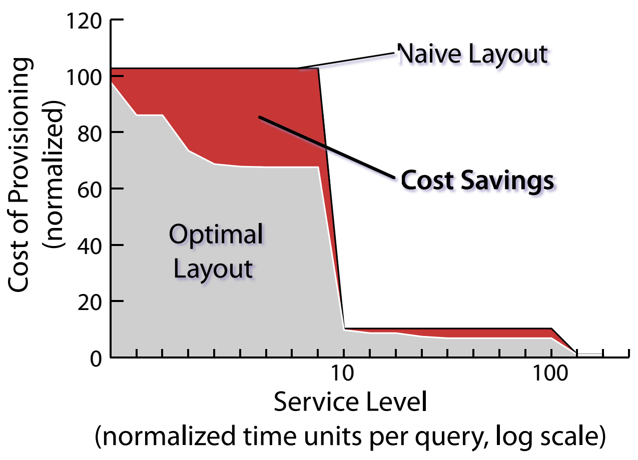

Figure 2: Costs of provisioning managed storage as a function of workload response time requirements when storage is disk-bound (See Section). The cost savings from choosing the optimal strategy over the naive one range up to 35 percent.

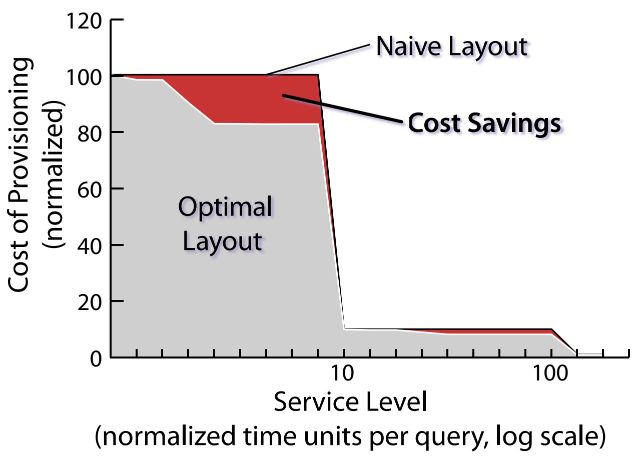

Figure 3: Costs of provisioning managed storage when the storage system is network-bound. The savings available through flexible use of QoS levels, while less than in Figure 2, are still significant.

We first computed the minimum and maximum possible response times for the query workload. These times occur when every data structure is mapped to the fastest or slowest storage service level, respectively. We then chose 20 response time thresholds between these two extremes. For each response time requirement, we computed the dollar cost of meeting the requirement by placing every data structure at the same storage service level. We refer to this first cost as the *naive cost* of meeting the response time requirement. Finally, we applied the OPBDP solver [4] to the constraint formulation in Section 5.3 to compute the mapping from data structures to storage devices that meets the response time requirement while minimizing total dollar cost. We refer to this second dollar cost as the *optimal cost* of meeting the response time requirement.

7 Experimental Results

7.1 Results with Storage Only Cost Structure

Our first cost structure simulated a storage system whose performance was limited by disk bandwidth. In this model, the storage system increases QoS levels by replicating or striping data or by using faster drives. As such, we assumed that the cost per megabyte of data stored in this model is proportional to the bandwidth that the storage must deliver.

Figure 2 shows a graph of the cost of provisioning managed storage under this cost structure and varying query response time thresholds. The line labeled “naive layout” shows the dollar cost of provisioning storage to meet the response time requirement when all data structures have the same quality of service. As the response time requirements become less stringent, the naive layout strategy shifts the entire database to lower QoS levels at once, resulting in large jumps in dollar cost.

The line labeled “optimal layout” shows the dollar cost of provisioning storage when each data structure can have any quality of service. Because the optimal layout places only the more performance-critical data structures on high QoS storage, the dollar cost curve for the optimal layout is smoother and dips more quickly as response time requirements are relaxed. Cost savings range up to 35 percent, with the greatest savings occurring when the naive strategy provides response times significantly less than what is required.

7.2 Results with Network Only Cost Structure

Our second cost structure simulated a storage system in which network bandwidth is the major performance constraint. In particular, we set the network cost of each service level to be proportional to the bandwidth the level provides.

Figure 3 shows a graph of the cost of managed storage as a function of the maximum average query response time. According to this graph, savings attainable by selective use of high QoS levels are about half as much under the network only cost structure as they are under a storage only cost structure. As before, cost savings were maximized when the naive strategy overshot the desired response time.

7.3 Results with Mixed Cost Structure

Our third cost structure combines the storage and network costs of the first two cost schemes, weighting these

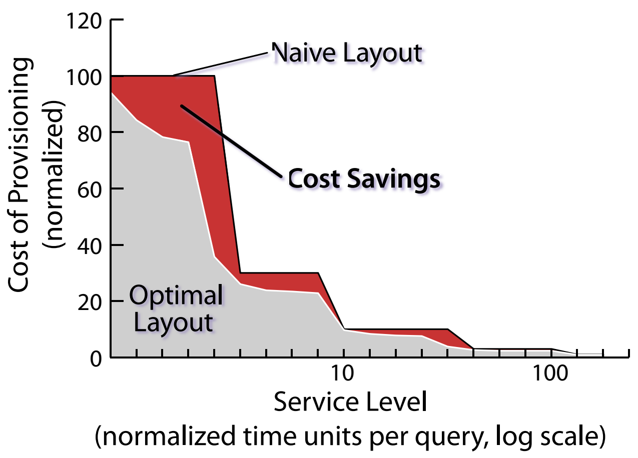

Figure 4: Costs of provisioning managed storage is equally network and disk bound. Results are in between those in Figure 2 and those in Figure 3.

cost components so as to make them approximately equal under the naive storage provisioning scheme. As shown in Figure 4, the results for this third cost structure were intermediate between the results for the first two cost structures.

7.4 Discussion

Our experiments demonstrate that database administrators can achieve significant cost savings through selective use of high storage QoS levels. In particular, storage systems whose performance is limited by disk bandwidth a are more amenable to optimization than those that are constrained by network bandwidth.

In the course of our experiments, we also observed an interesting phenomenon. In those cases when assigning all tables to the same service level causes the database to meet its response time requirements almost exactly, the cost of the naive layout comes close to the optimal cost. This effect may be an artifact of the TPC-H queries that we used for our experiment. The queries tend to exercise the different parts of the database equally. Hence, the value of fast access to a particular data structure in TPC-H is roughly proportional to the size of the data structure, and the naive provisioning strategy works well. Real-world query workloads tend to show more skew in their access patterns than does TPC-H. This skew should increase the benefits of selective use of high QoS levels.

8 Conclusion and Future Work

In this paper, we have studied the potential for data management service providers to save money through judicious use of storage quality of service. We have developed a theoretical model for reasoning about the problem. Using this model, we have given a rigorous definition of the problem of allocating out-of-core data structures to service levels so as to minimize dollar cost while meeting query response-time requirements. We have demonstrated how to convert this problem formulation into a Boolean quadratic program and then into a Boolean linear program. Finally, we have applied publically-available software to this linear program to demonstrate the potential for significant cost savings for a typical decision support workload on several classes of storage system.

Our experimental results lead us to conclude that, as IT outsourcing grows, service providers will benefit from having a software tool that maps portions of the database onto different QoS levels. Future database systems may even select storage service levels transparently without human intervention.

One issue that implementors of such a tool will face is that of scalability. The solver that we used in our experiments worked well for the scale of problem that we used in this paper. However, systems that provision storage for a significantly larger number of more-complex queries will benefit from using a more efficient optimization package. We are currently investigating the use of quadratic programming toolkits and approximation algorithms like simulated annealing and genetic programming as more scalable solutions to the storage provisioning problem.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Computer Systems*, 19(4):483–518, 2001.

[2] E. Anderson, R. Swaminathan, A. Veitch, G. Alvarez, and J. Wilkes. Selecting raid levels for disk arrays. In *FAST*, 2002.

[3] L. L. Ashton, E. A. Baker, A. J. Bariska, R. L. F. E. M. Dawson, S. M. Kissinger, T. A. Menendez, S. Shyam, J. P. Strickland, D. K. Thompson, G. R. Wilcock, , and M. W. Wood. Two decades of policy-based storage management for the IBM mainframe computer. *IBM Systems Journal*, April 2003.

[4] P. Barth. Opbdp. `http://www.mpi-sb.mpg.de/~barth/opbdp/opbdp.html`.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL). `http://www.w3.org/TR/wsdl`.

[6] I. B. M. Corporation. Db2 universal database version 8.1 for linux.

[7] I. B. M. Corporation. TPC benchmark H full disclosure report: IBM eServer xSeries 350 using IBM DB2 Universal Database 7.2. Technical report,

TPP Council, 2002. *http://www.tpc.org/results/FDR/tpch /x350_100GB_16proc_FDR.pdf*.

[8] T. P. P. Council. TPC Benchmark H. Technical report, Transaction Processing Performance Council, 2002. `http://www.tpc.org/tpch/spec/tpch150.pdf`.

[9] A. De Waegenaere and J. Wielhouwer. A partial ranking algorithm for resource allocation problems. Technical Report 2001-40, CentER, Tilburg University, The Netherlands, 2001.

[10] C. R. Dyer, A. Rosenfeld, and H. Samet. Region representation: Boundary codes from quadtrees. *Communications of the ACM*, 23(3):171–178, 1980.

[11] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13:91–128, 1988.

[12] S. Ganguly. Design and analysis of parametric query optimization algorithms. In *Proceedings of the Very Large Database Conference*, pages 228–238, 1998.

[13] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. Query optimization in the ibm db2 family. *Data Engineering Bulletin*, 16(4):4–18, 1993.

[14] GRAM. Grid resource allocation and mangement (GRAM). `http://www-unix.globus.org/toolkit/docs/3.2/gram/ws/`.

[15] HP. IT service management (ITSM). Technical report, 2005. `http://h20219.www2.hp.com/services/cache/10309-0-0-225-121.html`.

[16] A. Hulgeri and S. Sudarshan. Parametric query optimization. In *Proceedings of the Very Large Database Conference*, 2002.

[17] IBM. IBM global services — strategic outsourcing. Technical report, 2005. `http://www.ibm.com/services/be/so/`.

[18] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query processing. In *Proceedings of the Very Large Database Conference*, 1992.

[19] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 131–144, 2003.

[20] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM storage tank – a heterogeneous scalable SAN file system. *IBM Systems Journal*, July 2003.

[21] G. L. Nemhauser and L. A. Wosley. *Integer and Combinatorial Optimization*. Wiley, 1988.

[22] NEOS. `http://www-neos.mcs.anl.gov`.

[23] OGSA. Open Grid Services Architecture. `http://www.globus.org/ogsa/`.

[24] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proceedings of the ACM SIGMOD Conference*, 2002.

[25] F. Reiss and T. Kanungo. A characterization of the sensitivity of query optimization to storage access cost parameters. In *Proceedings of the ACM SIGMOD Conference*, 2003.

[26] S. Rozen and D. Shasha. A framework for automating physical database design. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the Very Large Database Conference*, pages 401–411. Morgan Kaufmann, 1991.

[27] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In P. A. Bernstein, editor, *Proceedings of the ACM SIGMOD Conference*, pages 23–34. ACM, 1979.

[28] SRM working group. The storage resource manager interface specification. `http://sdm.lbl.gov/srm-wg/doc/SRM.spec.v2.1.1.html`.

[29] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997.

[30] S. Uttamchandani, K. Voruganti, S. M. Srinivasan, J. Palmer, and D. Pease. Polus: Growing storage qos management beyond a "4-year old kid". In *FAST*, 2004.

A Linearization of the Quadratic Binary Optimization Problem

In this section we show how we map our the quadratic problem into a linear integer programming problem in binary variables. Without loss of generality, we will use a simpler notation to illustrate the process.

Let the binary quadratic objective function be $\sum_i \sum_j q_{ij} x_i x_j$ where $x_i, x_j \in \{0,1\}$. We define variables y_{ij} to represent $x_i x_j$. The new problem becomes:

$$\text{Maximize} \qquad \sum_{i=1}^{n} \sum_{j=1}^{n} q_{ij} y_{ij}$$

Subject to

$$\begin{aligned} y_{ij} &\leq x_i \text{ for all } i \text{ and } j \\ y_{ij} &\leq x_j \text{ for all } i \text{ and } j \\ y_{ij} &\geq x_i + x_j - 1 \text{ for all } i \text{ and } j \\ y_{ij}, x_i &\in \{0,1\} \text{ for all } i \text{ and } j \end{aligned}$$

First two constraints ensure that y_{ij} must be zero if either of x_i or x_j is zero. The next constraint ensures that y_{ij} is one if both are one. The rest constrain the variables to be binary.

The above system can be solved using branch-and-bound algorithms [21]. We used the OPBDP solver [4] but any appropriate solver could have been used (see the NEOS website [22]).