Workload-Aware Storage Layout for Database Systems

Oguzhan Ozmen, Kenneth Salem
University of Waterloo
Waterloo, ON Canada
{oozmen, kmsalem}@uwaterloo.ca

Jiri Schindler, Steve Daniel
NetApp, Inc.
{First.Last}@netapp.com

ABSTRACT

The performance of a database system depends strongly on the layout of database objects, such as indexes or tables, onto the underlying storage devices. A good layout will both balance the I/O workload generated by the database system and avoid the performance-degrading interference that can occur when concurrently accessed objects are stored on the same volume. In current practice, layout is typically guided by heuristics and rules of thumb, such as separating indexes and tables or striping all objects across all of the available storage devices. However, these guidelines may give poor results. In this paper, we address the problem of generating an optimized layout of a given set of database objects. Our layout optimizer goes beyond generic guidelines by making use of a description of the database system's I/O activity. We formulate the layout problem as a non-linear programming (NLP) problem and use the I/O description as input to an NLP solver. Our layout optimization technique, which is incorporated into a database layout advisor, identifies a layout that both balances load and avoids interference. We evaluate experimentally the efficacy of our approach and demonstrate that it can quickly identify non-trivial optimized layouts.

Categories and Subject Descriptors

H.2 [**Database Management**]: Physical Design

General Terms

Performance

1. INTRODUCTION

Database management systems (DBMS) rely on an underlying storage system for persistent storage of database objects such as tables, indexes, and logs. The storage system typically provides a set of RAID groups (groups of storage devices in some kind of RAID configuration) or individual storage devices, such as disk drives or solid-state drives (SSDs). We will refer to these as *storage targets*, by which we mean independent containers into which data can be stored. In this paper, we address the problem of laying out, or mapping, database objects onto these targets.

A good layout should result in a *balanced load* across the storage targets, otherwise, the most heavily loaded target may become a performance bottleneck while storage bandwidth available from other targets is wasted. It should also minimize *interference* between requests for different objects that are laid out on the same target. For example, if two sequentially-accessed database objects are laid out on the same target, interference among I/O requests for those two objects may prevent the underlying storage system from exploiting the sequentiality, thus increasing I/O service times [2, 25]. Finally, a good layout should reflect the *different characteristics* of the available targets. For example, RAID groups may vary in configuration, e.g., in the RAID level used, or in the number of devices in the group, and hence in performance. Heterogeneity may arise as new storage devices are added to storage systems over time, since the more-recently added devices are likely to be larger and faster. Heterogeneity may also occur by design. It is not uncommon to have mixed storage systems containing both enterprise-class 15K RPM disk drives targeted for high throughput of small random I/O requests and cost-effective nearline 7200 RPM disks, better suited for sequential accesses. Similarly, a flash memory SSD storage target will have much better random I/O performance than one implemented using disk drives. The promise of good performance for database workloads with SSDs [13] suggests that storage systems will likely continue to include heterogeneous configurations with SSDs or high-end disks for small random I/Os and cost-effective disk drives for large sequential accesses. A layout that fails to account for these kinds of heterogeneity may result in poor performance.

In current practice, layout is based primarily on heuristics and rules-of-thumb. For example, one strategy is to distribute every database object across all available targets, an approach that is known as *stripe-everything-everywhere (SEE)* [18, 22]. For homogeneous storage systems this may indeed be a good way to balance load, but it ignores the interference problem and it is not clear how to apply it effectively in the presence of heterogeneous targets. Another strategy is to lay out tables, indexes, and logs on separate targets in order to reduce interference among them. However, this leaves unresolved the question of which target(s) to use for each type of object, and poor choices may lead to lower performance due to load imbalances.

In this paper we propose a technique for identifying optimized database layouts automatically. Our technique, which

SIGMOD'10 June 6–11, 2010, Indianapolis, Indiana, USA.

could be implemented as a standalone *database storage layout advisor*, makes use of information about the I/O access patterns of each database object and employs storage target models that allow it to recognize the targets' performance characteristics. This paper makes the following contributions:

- We formulate the database storage layout problem as a non-linear optimization problem incorporating the important characteristics of the storage workload and of the underlying storage targets.
- We propose a technique for solving the layout problem to identify good layouts. Our technique exploits a generic non-linear program (NLP) solver as well as heuristics specific to the layout problem.
- We present an experimental evaluation of the efficacy and efficiency of our technique.

The rest of the paper is structured as follows. Section 2 presents a scenario which illustrates the effect of object layout on DBMS performance. Section 3 defines the layout problem and Sections 4 and 5 describe our technique for layout optimization. Section 6 presents an empirical evaluation of the technique's efficacy and cost. Section 7 summarizes related work on the layout problem, and Section 8 concludes and briefly discusses some future directions for this work.

2. LAYOUT EXAMPLE

Before we present our layout advisor, we illustrate workload-aware database layout with an example of a PostgreSQL database system managing a small TPC-H [8] database. Our system has four identical storage targets, each of which is a single disk drive. The TPC-H database has 20 different objects, including tables, indexes, and a tablespace for temporary objects, that must be laid out on the four targets. The PostgreSQL database runs our OLAP1-63 workload, which consists of a sequence of TPC-H benchmark queries. Section 6 provides more detailed descriptions of both the OLAP1-63 workload and the system.

Figure 1 shows two different layouts of the database objects across the four storage targets. For clarity, we only show the layouts of the eight most heavily accessed objects, which include four tables, three indexes two of which are defined on the `LINEITEM` table and the other on the `ORDERS` table, and the tablespace for temporary objects. The illustration on the left shows the stripe-everything-everywhere (SEE) layout, which distributes each object evenly across all four of the disks. The illustration on the right shows the layout that is recommended by our layout advisor based on the I/O workloads of each of the objects and the performance characteristics of the devices.

In the optimized layout, the two most heavily accessed tables, `LINEITEM` and `ORDERS`, are isolated from one another. Both tables exhibit sequential access patterns, and they tend to be accessed concurrently, so isolating these tables from one another helps to avoid interference. `LINEITEM` is laid out on more targets than `ORDERS` because `LINEITEM` experiences a greater I/O load. The next most heavily accessed object, `I_L_ORDERKEY`, is also isolated from both `LINEITEM` and `ORDERS` to avoid interference with those objects' sequential access patterns. The `TEMP SPACE` (temporary table space) object also experiences a sequential workload. Had more storage targets been available, the layout optimizer may have

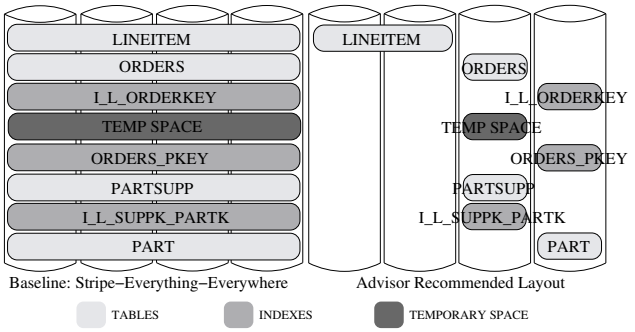

Figure 1: SEE and Optimized Layouts of TPC-H Database Objects.

chosen to isolate it as well. Since this is not possible, `TEMP SPACE` is placed on the same target as `ORDERS` as these two objects are rarely accessed simultaneously in this workload. The remaining objects, which experience low I/O loads, are placed on the least heavily loaded targets.

On our system, the OLAP1-63 workload under the SEE layout completes in 40927 seconds. Under the optimized layout, the same workload completes in 31879 seconds, a 1.28x speedup. While this reduction may not seem dramatic, the scenario we have constructed is very friendly to SEE since the available storage targets are homogeneous. Even in this environment, an optimized layout can improve performance. The benefits of an automatic, workload-aware layout advisor are more significant in situations for which SEE or other similar heuristics are poorly suited, for example, when the storage targets are heterogeneous. Section 6 shows that the benefits of layout optimization increase in these cases.

3. PROBLEM FORMULATION

We assume that the storage system provides M disjoint storage targets, and we use c_j to denote the capacity of the jth target. The exact nature of the targets is not important to our layout advisor. It cares only that each storage target has an associated performance model (as we will describe shortly) and that the performance of each storage target is independent of the performance of the other targets. In an enterprise-class storage system [10, 12, 17], a target might correspond to a RAID group, i.e., a set of storage devices in a particular RAID configuration. In smaller-scale settings, an individual storage device directly attached to a server could be a storage target.

The layout advisor is given N disjoint database objects that are to be laid out on the available storage targets. We use s_i to denote the size of the ith object. Again, the exact nature of the database objects is not important. They could be tablespaces, individual tables, indexes, or logs. All of the objects may be part of the same database or they may originate from different databases.

A *layout* L is an $N \times M$ matrix, where $0 \leq L_{ij} \leq 1$ represents the fraction of object $object_i$ that is assigned to target $target_j$. We are interested only in *valid* layouts, which are layouts that satisfy two additional constraints:

capacity constraint: $\sum_{i=1}^{N} s_i L_{ij} \leq c_j. \qquad \forall j, 1 \leq j \leq M$

integrity constraint: $\sum_{j=1}^{M} L_{ij} = 1. \qquad \forall i, 1 \leq i \leq N$

The capacity constraint ensures that a target is not assigned more data than it can hold, and the integrity constraint

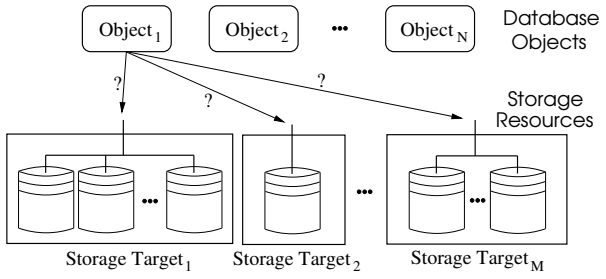

Figure 2: Laying out Objects onto Storage Targets.

Symbol	Description
N	Number of database objects
s_i	Size of ith object
M	Number of storage targets
c_i	Capacity of ith target
W_i	I/O workload for the ith object
μ_j	Utilization of the jth storage target

Figure 3: Summary of Layout Problem Parameters

ensures that each object is mapped in its entirety to the storage target(s). Figure 2 illustrates the layout of database objects onto storage targets.

The goal of the layout advisor is to identify a good layout from among all possible valid ones. Specifically, the objective is to minimize the maximum *utilization* across all storage targets. This objective encourages the advisor to identify balanced layouts, since the most heavily utilized target determines the quality of the layout. It also encourages the advisor to avoid interference, since interference increases I/O request service times and hence storage target utilizations. Finally, it encourages the advisor to consider the performance characteristics of the targets as it makes layout decisions. For example, with a mix of fast and slow storage targets in the system, good layouts will tend to place more load on the faster targets than on the slower ones.

To determine the storage target utilizations resulting from a candidate layout, the advisor needs information about the I/O workload. We assume that the layout advisor is given a *workload description* for each database object, and we use W_i to denote the workload description for the ith object. An object's workload description characterizes the the object's I/O activity. For example, it describes the I/O request rate for the object, the mix of reads and writes, the sequentiality of the requests, and other properties. Section 5.1 describes the I/O workload model in more detail.

To predict storage target utilizations for a given workload, the layout advisor also needs a performance model for each target. We use $\mu_j(W_1, \ldots, W_N, L)$ to denote the model-predicted utilization of the jth storage target under the I/O workloads $W_1, \ldots, W_N$ and layout L. To simplify our notation, we will refer to the utilization of the jth target as μ_j when the workloads and layout on which it depends are clear from the context. We defer further discussion of performance models and the calculation of μ_j to Section 5.2. Figure 3 summarizes the layout problem parameters.

We can now state the layout problem:

DEFINITION 1. ***Database Object Layout Problem***
Given N database objects, M storage targets, and an I/O workload description W_i for each object, find a valid layout L that minimizes

$$\max_{j=1}^{M} \mu_j(W_1, \ldots, W_N, L)$$

Once the layout advisor has recommended an optimized layout, a variety of mechanisms can be used to implement the layout. Most storage systems provide mechanisms for defining *logical volumes*, which are mapped to underlying storage targets. A recommended layout L can be implemented by defining logical volumes for database objects, or for groups of database objects that have the same layout in L. The storage system's mechanism for mapping logical volumes to the underlying RAID groups or devices can then be used to implement L. On the host side, operating system supported logical volume managers (LVMs) can be used to implement a layout in essentially the same way. Finally, at the application level, many database systems are capable of distributing database objects (e.g., tablespaces) across containers (e.g., files or raw devices) provided by the underlying operating system [18]. By defining one or more containers for each object and placing them appropriately on the available storage targets, a database administrator can use this mechanism to implement the recommended layout.

Some layout mechanisms are more limited than others in the types of layouts that they can support. For example, some mechanisms use round-robin striping of objects. However, this always distributes an object evenly across some set of targets. A layout that maps, for example, 20% of an object to one target, 30% to a second target, and 50% to a third target is a valid mapping according to our definitions, but it cannot be implemented by such a mechanism. For this reason, we provide the ability to optionally restrict the layout advisor to *regular* layouts:

DEFINITION 2. ***Regular Layout***
A regular layout is a valid layout in which, for every pair of elements L_{ij} and L_{ik}, either $L_{ij} = 0$ or $L_{ik} = 0$ or $L_{ij} = L_{ik}$.

In a regular layout, each object is distributed evenly across one or more storage targets. For example, 50% of the object may be placed on each of two targets, or 25% may be placed on each of four targets.

4. LAYOUT OPTIMIZATION

The two key elements of our technique for solving the layout problem are (i) its formulation as a non-linear programming (NLP) problem and (ii) the use of a generic NLP solver. By using a generic solver, we can directly leverage the solver's techniques for exploring the optimization space. Explicitly formulating layout as a non-linear programming problem makes it easy to incorporate additional constraints on the resulting layout. For example, if administrative constraints require certain objects to be laid out onto particular targets, we can easily add such constraints to the NLP problem before solving it. As discussed in Section 6, we also found the generic solvers to be fast and effective.x

4.1 Solver

We use AMPL [11], a standard mathematical modeling language, to formulate the layout problem as a non-convex NLP. There exist several general solvers designed to solve

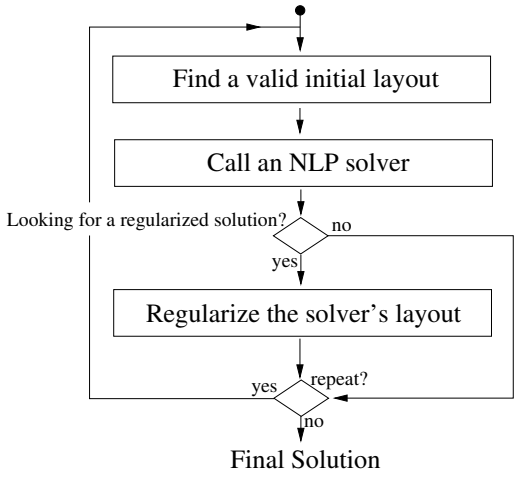

Figure 4: Layout Algorithm.

non-convex optimization problems with non-linear objective functions and non-linear constraints. We use the MINOS solver [16], because it supports the use of external (i.e., non-AMPL), black-box functions as part of the definition of the objective function. We leverage this feature for our performance models as described in Section 5.

In order to use a generic NLP solver like MINOS, we must address several issues. First, we need to provide a valid initial layout. This step is described in more detail in Section 4.2. Second, the solver will find a valid, optimized layout, but the resulting layout may not be regular. If a regular layout is required, then we need a way to *regularize* the solver's solution. This regularization step is described in Section 4.3. Figure 4 summarizes the behaviour of our layout optimizer.

Many NLP solvers, like MINOS, are not guaranteed to identify a globally optimal solution. Moreover, the solution found by the solver may depend on the choice of the initial layout. Thus, it is possible to improve the quality of the optimized layout, at the cost of additional optimization time, by repeating the optimization process using different initial layouts. The optimization procedure described in Figure 4 incorporates this iteration. A potential advantage of using multiple initial layouts is that they offer a convenient way of introducing the knowledge of domain experts into the optimization process. For example, if a knowledgeable database administrator expects that certain layouts might perform well, those layouts can be used as initial layouts. We have not yet explored this possibility in our layout advisor; all of the results presented in Section 6 use a single initial layout. The choice of a suitable initial layout is detailed below.

4.2 Initial Layout

Originally, we experimented with SEE as the initial layout as it is simple and balanced. However, it often represents a local minimum in the search space, and we found that MINOS at times had difficulty breaking out of this minimum. Therefore, SEE is not an ideal choice if the layout advisor uses only a single initial layout. To avoid this problem, we switched to a simple heuristic for choosing an initial layout: placing database objects based on their sizes and total request rates. An object's total request rate is one of its workload characteristics, which we describe in more detail in Section 5.

The initial configuration is chosen by laying out one object at a time, in decreasing order of request rate. Each object is assigned to the target that has the lowest total assigned request rate from among those targets that have sufficient remaining storage capacity to hold the object. This approach assigns each object to a single storage target. By assigning each object to the least-loaded target, we hope to obtain an initial layout that is at least approximately balanced. However, the heuristic does not make any attempt to minimize interference, nor does it take into account the performance characteristics of the storage targets.

4.3 Regularization

If non-regular layouts can be handled by the storage system, operating system or database system that is responsible for implementing them, then the layout generated by the NLP solver can be implemented directly and there is no need for regularization. For systems that only support regular layouts, the layout advisor needs to perform the final regularization step.

One way to generate regular layouts would be to add regularization constraints directly to the NLP problem formulation. However, such constraints would effectively turn a continuous optimization problem (each of the decision variables L_{ij} is a continuous variable in the range $0 \leq L_{ij} \leq 1$) into a combinatorial problem. With the addition of regularization constraints, there are up to $2^M - 1$ possible layouts for each object, and hence $O(2^{MN})$ possible layouts for all objects. To solve the regularized problem effectively, we should use a solver that is intended for combinatorial problems. Instead, we use the solver to identify an optimized non-regular layout and then apply a post-processing step to regularize the optimized layout.

The regularization algorithm starts with the non-regular optimized layout produced by the solver and regularizes the layout of one object at a time, until all objects' layouts are regular. The algorithm regularizes the objects in decreasing order of the total storage system load (for object i, $\sum_j \mu_{ij}$) they impose on the storage system. By considering the objects in this order, load imbalances introduced by regularizing the initial objects can potentially be corrected by the regularizations of subsequent objects. Load imbalances created by regularization of the final objects will be uncorrectable but relatively small.

To choose a regular layout for a particular object, the algorithm generates a small set of candidate regular layouts. Two classes of candidates are generated. The first class consists of all regular layouts of the object that are *consistent* with the layout chosen by the solver. By consistent, we mean that if $L_{ij} > L_{ik}$ in the original layout generated by the solver, then only regular layouts for which $L_{ij} \geq L_{ik}$ will be candidates. For example, if there are 3 storage targets and the solver lays out an object as follows: $(47\%, 35\%, 18\%)$, then the regularization algorithm will consider only the following regular layouts: $(100\%, 0\%, 0\%)$, $(50\%, 50\%, 0\%)$, and $(33\%, 33\%, 33\%)$.[1] The second class of candidates consists of regular layouts that place the object on the least loaded targets, according to the current layout. Specifically, the regularizer considers layouts that

[1]In case $L_{ij} = L_{jk}$ in the solver's layout, the tie is broken arbitrarily, using target identifiers.

Parameter	Symbol	Description
Request Size	B_i^R/B_i^W	Avg. read/write request sizes
Request Rate	λ_i^R/λ_i^W	Avg. read/write request rates
Run Count	Q_i	Avg. number of requests in a sequential run
Overlap	$\mathcal{O}_i[j]$	Temporal correlation of I/O requests in this workload with I/O requests from the jth workload.

Figure 5: Parameters of ith Object's Workload (W_i)

assign 100% of the object to the least loaded storage target, 50% of the object to each of the two least loaded targets, and so on. We call these *balancing* layouts, since they tend to correct load imbalances that may arise from the regularization of previous objects. For each object, there will be $2M$ candidate regular layouts, M of each class. From this set of candidates, the regularizer eliminates any layouts that would result in violations of the targets' space constraints, and from the remaining valid layouts it chooses the one that minimizes the optimization objective, i.e., the regular layout that minimizes the maximum utilization of the storage targets.

It is possible that, as a result of space constraints, all of the regular layouts considered by the regularization algorithm for some object will be invalid. In this case, the regularization algorithm may fail to generate a regular layout (even if such a layout does exist), and manual intervention would be necessary. In practice, this is unlikely to be a problem unless space constraints are very tight.

5. WORKLOAD AND TARGET MODELS

As described in Section 3, the layout advisor relies on workload descriptions and storage target performance models. We use models that are based on ideas that were developed at HP Laboratories, and which were used there as the basis of a series of storage management tools, such as Minerva [3] and Ergastulum [5]. We call them "Rome-style" models as they use the Rome language [28] to describe workloads and storage devices. For our purposes, we need to ensure that our Rome-style models are compatible with the NLP solver.

5.1 Workload Model

The layout advisor's input includes a set of workload descriptions, W_i, one for each database object. Each workload is modeled as a stream (sequence) of block I/O requests. The requests in the stream are characterized by a set of parameters summarized in Figure 5. The values of these parameters constitute the workload description. The run count parameter describes the sequentiality of the workload. It describes the number of sequential requests that occur between non-sequential jumps, so higher run counts indicate sequential workloads. The Overlap parameters characterize the level of temporal overlap between the workload's I/O requests and the I/O requests of other workloads. The value of the Overlap parameters ranges within [0,1], with $\mathcal{O}_i[j] = 0$ meaning that requests from W_i never overlap with those of W_j and $\mathcal{O}_i[j] = 1$ indicating that they always do. Overlapping streams are potentially subject to interference if they are laid out on the same storage targets.

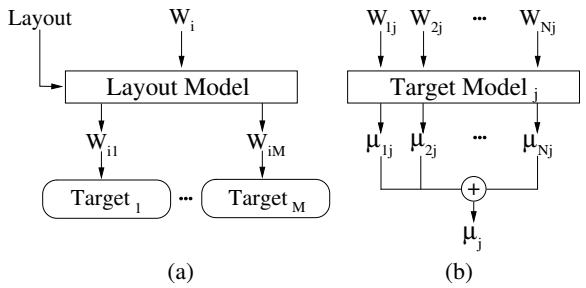

Figure 6: Storage System Model

There are several ways to obtain these workload descriptions. We obtained them by collecting a trace of I/O activity from the operational database system. We then isolate the requests related to a single object from that trace and determine values of the workload parameters for that object by fitting them to the observed characteristics of the traced requests. We used a trace analysis tool called Rubicon [26] to do the parameter fitting.

Another possibility is to directly infer the storage workload descriptions using knowledge of the database system and its workload and a tool called a *storage workload estimator* [19]. This allows storage workload descriptions to be generated without actually running the workload and collecting traces. However, the resulting descriptions may be less accurate than those obtained using the trace-based method.

5.2 Storage System Model

The layout advisor requires a model that can predict the utilization of a storage target, given descriptions of the workloads and a candidate layout. In theory, any function that maps workloads and layout to utilization could be used. In practice, the models we use have a particular internal structure, as illustrated in Figure 6. These models define the NLP solver's objective function.

As depicted in Figure 6(a), a *layout model* takes as an input an object's workload, W_i, and the candidate layout, L, and produces a description of the workload that will be generated by that object on each storage target. We use W_{ij} to denote the workload imposed by the ith object on the jth storage target under a given layout. The layout model, which is described in more detail in Section 5.2.1 describes the distribution of the object's workload that is implied by the candidate layout.

The advisor applies the layout model to each object's workload. As a result, each storage target has a set of incoming workloads, one from each object, as illustrated in Figure 6(b). Given these workloads, a *target model*, which is described in more detail in Section 5.2.2, is used to produce an estimate of the storage target utilization attributable to each input workload. There may be a different model for each target type or even for each individual target. The storage target's total utilization, which the solver seeks to minimize, is the sum of those loads. Recall that μ_{ij} denotes the utilization of the jth target that is attributable to the ith object, and μ_j represents the jth target's total utilization.

One advantage of this structure is that the layout model is relatively simple and can be readily expressed in AMPL. This exposes the characteristics of this part of the model to the NLP solver. The target model, on the other hand, must

$$
\begin{aligned}
B_{ij}^R &= B_i^R \\
B_{ij}^W &= B_i^W \\
\lambda_{ij}^R &= \lambda_i^R L_{ij} \\
\lambda_{ij}^W &= \lambda_i^W L_{ij} \\
Q_{ij} &= \begin{cases} Q_i & \text{if } Q_i B_i < \text{StripeSize} \\ Q_i L_{ij} & \text{if } Q_i B_i > \text{StripeSize}/L_{ij} \\ \text{StripeSize}/B_i & \text{otherwise} \end{cases} \\
\mathcal{O}_{ij}[k] &= \begin{cases} \mathcal{O}_i[k] & \text{if } L_{ij} > 0 \text{ and } L_{kj} > 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 7: Layout Model for LVM Using Striping. StripeSize is the LVM stripe size, and B_i is the request-rate-weighted average of B_i^R and B_i^W.

capture the potentially complex performance characteristics of the target. These models are not expressed in AMPL, but rather as external functions invoked by the MINOS solver. This allows us to "plug in" models for different targets without changing the AMPL formulation of the rest of the NLP problem. The remainder of this section describes the layout and target models in more detail.

5.2.1 Layout Model

The layout model describes how to transform an object workload into per-target workloads, given a particular layout. The per-target workload descriptions W_{ij} are of the same form as the original workload descriptions W_i (Figure 5). Thus, a layout model describes how to calculate each of the parameters (e.g., RunCount, Request Rate) of W_{ij} from the parameters of W_i and the layout.

The layout model must capture the effects of whatever system is responsible for implementing the layout. This may be a RAID controller, a logical volume manager in the host operating system, or an underlying storage system. For the experiments reported in Section 6, layouts were implemented by the logical volume manager (LVM) in the host operating system. As configured, the LVM divides objects into fixed-size stripes and distributes the stripes round-robin to the underlying storage targets. Our layout model for this LVM is shown in Figure 7.

5.2.2 Target Model

A target model estimates the storage target utilization imposed by each workload (μ_{ij}), the sum of which is the total utilization of the storage target (μ_j). We construct two request cost models for each type of target, one model for read requests and the other for write requests. We then estimate μ_{ij} using

$$\mu_{ij} = \lambda_{ij}^R \text{Cost}_j^R + \lambda_{ij}^W \text{Cost}_j^W \quad (1)$$

where Cost_j^R and Cost_j^W are the read and write request costs for the jth storage target.

The read and write request costs for workload W_{ij} depend on the characteristics of the jth target storage device and the properties (request sizes, run count) of the workload. Furthermore, because of the potential for interference among workloads, the request costs also depend on the other workloads W_{kj} that impinge on the same target. Thus, in general the cost model will depend on a large number of factors. To eliminate some of this complexity, we assume that the amount of interference with the requests from workload W_{ij} depends on the *number* of competing requests from other workloads, and not on the properties (request size, sequentiality) of those competing requests. This allows us to compute, for each workload W_{ij}, a *contention factor*, χ_{ij}, which captures the total amount of interference from all other workloads on target j:

$$\chi_{ij} = \frac{\sum_{k \neq i} (\lambda_{kj}^R + \lambda_{kj}^W) \mathcal{O}_{ij}[k]}{(\lambda_{ij}^R + \lambda_{ij}^W)} \quad (2)$$

Essentially, χ_{ij} is a measure of the number of temporally-correlated requests from other workloads (W_{kj}) per request from workload W_{ij}.

With this simplification, the per-request costs Cost_j^R and Cost_j^W are functions of the properties of storage target j and three workload parameters: request size, run count (sequentiality), and the contention factor. To estimate these costs, we construct two cost models (one for reads, one for writes) for each type of device, parametrized by the three workload parameters. It is possible, but difficult, to build accurate analytic cost models [24, 25]. Instead, we construct black-box models based on interpolation among tabulated measurements of storage target performance. Others have used similar techniques to build black box storage device models [4, 15, 27]. We construct the models by subjecting the storage targets to calibration workloads with known request sizes, run counts, and degrees of contention and measuring the request service times, which are then tabulated. To estimate Cost_j^R or Cost_j^W given a set of workload parameter values, we look up the tabulated cost from the appropriate model, interpolating among nearby calibration points if necessary.

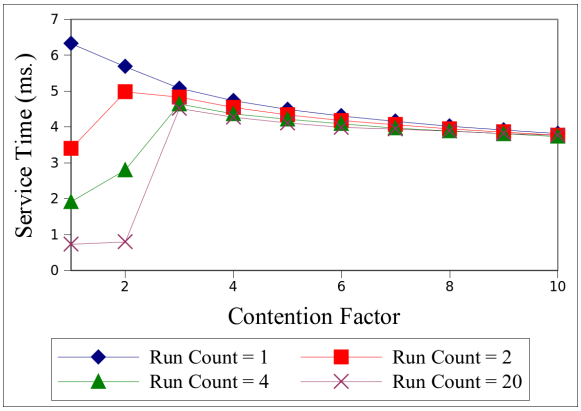

Figure 8: Cost Model for 8 KByte Read Requests

Although the behavior of storage devices can be complex and highly non-linear, the generality of the tabulation/interpolation approach allows us to model them accurately. Figure 8 shows one "slice" of the read request cost model for the SCSI disk drives used in our experiments. This slice corresponds to read requests of size 8 KBytes. The figure shows request costs as a function of the contention factor. Each curve corresponds to a different run count (degree of sequentiality) in the workload. This slice of the model illustrates several interesting effects. When contention is low, sequential requests are significantly faster than non-sequential ones, as expected. The sequential advantage is preserved in the face of a small amount of contention because the device is able to track and read-ahead on a small number of concurrent sequential streams. However, the advantage collapses

DB	Total Size	Number of Objects			
		Tables	Indexes	Temp Space	Log
TPC-H	9.4GB	8	11	1	0
TPC-C	9.1GB	9	10	0	1

Figure 9: Databases Used in the Experiments

Workload	Number of Queries	Concurrency Level	Target Database
OLAP1-21	21	1	TPC-H
OLAP1-63	63	1	TPC-H
OLAP8-63	63	8	TPC-H
OLTP	n/a	9	TPC-C

Figure 10: Query Workloads Used in the Experiments

quickly completely when the contention factor reaches 2. The cost of non-sequential requests (RunCount = 1) gradually *decreases* with increasing contention because disk head scheduling is more effective when there is a larger request queue.

6. EVALUATION

In this section, we present an experimental evaluation of the layout algorithm. Our primary goal is to evaluate the quality of the optimized layouts that are recommended by the layout advisor. In addition, we consider the time required by the layout advisor to produce a recommendation. Finally, we compare our NLP-based layout advisor to a previously-proposed technique [2] for laying out relational databases.

6.1 Experimental Setup

To evaluate the quality of the optimized layouts, we compare the performance of a database system that uses an optimized layout to the performance of the same database system using a *baseline* layout. The baseline layouts are simple heuristic layouts that make little or no use of workload information. The objects to be laid out are individual database tables, indexes, logs, and tablespaces for temporary objects. In most cases, our performance metric is the total elapsed (wall-clock) time required to execute a particular set of queries. In addition to the elapsed times, which are our primary metric, we also record the *estimated* storage target utilizations (μ_j) that are used internally by the layout advisor to judge the quality of layout. We use these estimated utilizations to illustrate and explain the advisor's behaviour.

In our experiments, we used the PostgreSQL database management system, version 8.0.6. We used two databases: a scale factor 5 TPC-H database and a scale factor 90 TPC-C database. The characteristics of the objects in these databases are shown in Figure 9. We define four SQL workloads running against these databases, as shown in Figure 10. The OLAP1-21 workload consists of 21 of the 22 TPC-H benchmark queries. (TPC-H query Q9 was excluded because of its excessive run-time in our system.) In OLAP1-21, the queries are executed sequentially in a randomly selected order, with no think times. OLAP1-63 is simply a longer variant of OLAP1-21 in which each TPC-H query occurs three times in the mix. The entire mix is randomly permuted and the queries are executed sequentially. OLAP8-63 is the same as OLAP1-63 except that queries are executed with a concurrency level of eight. That is, whenever a query finishes, the next query in the sequence is started so that eight queries are active at all times. The OLTP workload is generated by nine simulated terminals, with no think time or keying time.

Workload	Baseline (SEE) Execution Time (seconds)	Optimized Execution Time (seconds)	Speedup
OLAP1-63	40927	31879	1.28x
OLAP8-63	16201	13608	1.19x

Figure 11: Workload Execution Times for Baseline and Optimized Layouts on Homogeneous Storage Targets

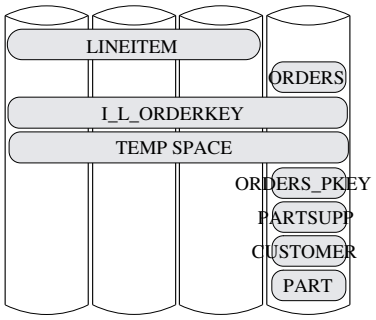

Figure 12: Optimized Layout for the OLAP8-63 Workload

We ran PostgreSQL on a Dell PowerEdge 2600 server with two 2.4GHz Intel Xeon processors and 4GB of main memory, running SUSE 10.0 Linux with kernel version 2.6.21.7. We instrumented the kernel so that we were able to obtain the I/O request traces that we used to build our workload models, as described in Section 5. The size of the database system's shared buffer is set to the maximum allowable value (2GB) for the three TPC-H workloads, and to 1.5GB for the OLTP workload. The server has a 70GB 15K RPM SCSI disk which holds all system software, including PostgreSQL itself. In addition, it has four 18.4GB 15K RPM SCSI hard drives behind a configurable Dell Perc 4Di RAID controller and a 32GB solid-state drive (SSD) behind a 3Gb/s Koutech SATA-II controller. We lay out the TPC-H and TPC-C database objects on various combinations of the four 18.4GB hard drives and the SSD in our experiments.

6.2 Layout Quality: Homogeneous Targets

In this experiment, we compare DBMS performance under advisor-recommended layouts with performance under a baseline stripe-everything-everywhere (SEE) layout. We used the OLAP1-63 and OLAP8-63 workloads. For each workload, the layout advisor is asked to recommend a layout of the TPC-H database objects onto four identical storage targets: the four 18.4GB disk drives attached to our server.

Figure 11 summarizes the results of this experiment. Although this scenario is well-suited to SEE, the optimizer is able to obtain some performance improvement for both workloads. In Figure 1 (Section 2) we illustrated the opti-

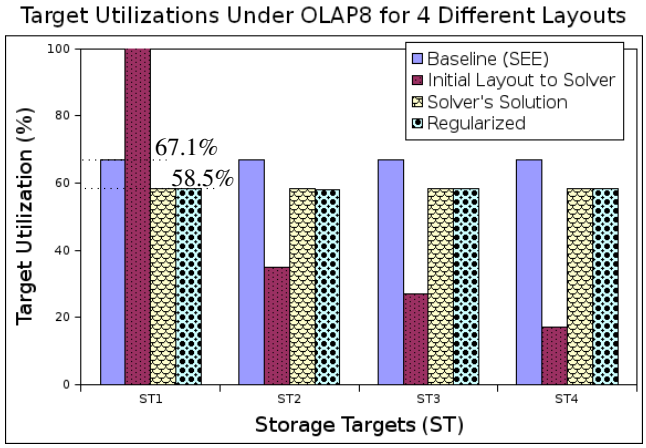

Figure 13: Estimated Utilizations Under the OLAP8-63 Workload

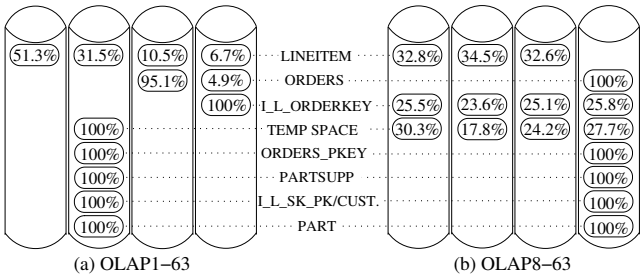

Figure 14: Layouts Produced by the NLP Solver Under the OLAP1-63 and OLAP8-63 Workloads

mized layout that the advisor recommends for the OLAP1-63 workload. Figure 12 shows the optimized layout for the OLAP8-63 workload. As was the case in Figure 1, the objects are shown in decreasing order of request rate, and only the most heavily-requested objects are shown. For both the OLAP8-63 workload and the OLAP1-63 workload, the recommended layouts separate the two most heavily-used objects (`LINEITEM` and `ORDERS`). For OLAP8-63, the workload on `LINEITEM` is less sequential than it is under OLAP1-63, because of query concurrency. As a result, the performance penalty for interference with `LINEITEM` is lower under the OLAP8-63 workload, and `LINEITEM` is not completely isolated in the OLAP8-63 layout. Instead, the optimizer distributes `I_L_ORDERKEY` and `TEMP SPACE` across all of the targets to better balance the load.

Figure 13 illustrates the behaviour of the layout advisor by showing the quality of the layouts that it considers at different stages of its execution. In the figure, each group of bars shows the estimated utilization (μ_j) of one of the four storage targets. The leftmost bar in each group shows the advisor's estimated utilization for the baseline SEE layout, for the purposes of comparison. The second bar in each group shows the estimated target utilization under the *initial* workload generated by the layout advisor as a starting point for the NLP solver. These initial layouts assign each object to a single storage target. They tend to be unbalanced, as is the case for both OLAP8-63 and OLAP1-63, because the advisor does not take interference, workload sequentiality, and other factors into account when generating them.

Workload	SEE Baseline Performance	Optimized Performance	Improvement Optimized vs. SEE
OLAP1-21	24416 sec.	17005 sec.	1.43x
OLTP	304 tpmC	360 tpmC	1.18x

Figure 15: Consolidation Scenario Performance Under Baseline and Optimized Layouts on Homogeneous Storage Targets

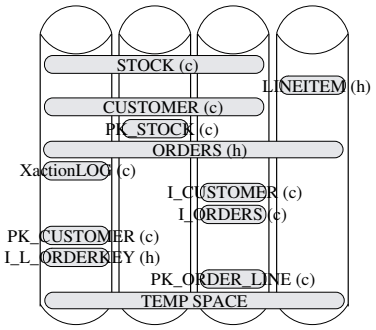

Figure 16: Optimized Layouts of the TPC-H (h) and TPC-C (c) Objects for the Consolidated Workload on Homogeneous Storage Targets

The third bar in each group shows the estimated utilization for the (non-regular) layout identified by the NLP solver. The solver's layouts for both OLAP1-63 and OLAP8-63 are shown in Figure 14. These layouts are very balanced and they reduce utilizations relative to the default SEE layout.

If the layout mechanism supports non-regular layouts, then the layouts shown in Figure 14 can be implemented directly. If not, then the final regularization step is needed. The fourth bar in each group in Figure 13 shows the estimated utilizations under the final regularized layout that the advisor produces in its post-processing step. In general, these layouts will be less balanced than those produced by the solver, as the regularization process disturbs the balanced layout produced by the solver, and it may be unable to completely correct these disturbances. However, in the case of the OLAP8-63 workload, the solver's layout (Figure 14(b)) is almost regular, and the resulting regularized layout (Figure 12) is very close to the solver's.

6.3 Layout Quality: Consolidation Scenario

In these experiments, we considered a consolidation scenario in which two database system instances run on the server. One instance runs our OLAP1-21 workload and the second runs our OLTP workload. In this experiment, we measure the performance of the OLAP1-21 workload by measuring the wall-clock time required to complete the workload queries. For the OLTP workload, we measure performance in TPC-C New-Order transactions per minute (tpmC). We run the OLTP workload until the OLAP1-21 workload finishes. The reported tpmC rate is the average rate over the lifetime of the experiment, minus an initial warm-up period of 1600 seconds.

In this scenario, there are a total of 40 database objects to be laid out, including 20 from the TPC-H database and

Storage Target Config	Baseline (SEE) Execution Time (seconds)	Baseline (isolate tables) Execution Time (seconds)	Baseline (isolate tables & indicies) Execution Time (seconds)	Optimized Execution Time (seconds)	Speedup (Optimized vs. SEE)
3-1	18103	14507	n/a	13317	1.36x
2-1-1	16922	n/a	22359	13163	1.29x
1-1-1-1	16201	n/a	n/a	13608	1.19x

Figure 17: Workload Execution Times for Baseline and Optimized Layouts on Heterogeneous Storage Targets, for the OLAP8-63 Workload

20 from the TPC-C database. We used the layout advisor to generate an optimized layout of the 40 objects onto the four 18.4GB disk drives. We compared this layout against a SEE baseline, which stripes all 40 objects across the four drives.

Figure 15 summarizes the results of this experiment, and Figure 16 shows the regular layout recommended by the advisor for the 12 most heavily-requested objects. Optimization boosts the performance of both the OLAP1-21 and OLTP workloads. This is achieved primarily by separating the TPC-H `LINEITEM` table from the TPC-C `STOCK` and `CUSTOMER` tables, which see heavy non-sequential workloads in our configuration.

6.4 Layout Quality: Heterogeneous Targets

In these experiments we asked the layout advisor to recommend layouts for heterogeneous storage targets. In the first experiment, we used our server's RAID controller to create heterogeneous targets out of the four 18.4GB disk drives. In the "3-1" configuration, we created a 3-disk RAID0 target from three of the disks and used the remaining disk as a standalone target. In the "2-1-1" configuration, we created a 2-disk RAID0 target and used each of the remaining two disks as standalone single-disk targets. In practice, heterogeneity may arise not only from storage system configuration, but also from the presence of multiple types of devices or storage systems. In a second experiment, we asked the layout advisor to recommend layouts across the four 18.4GB disk drives plus the SSD. We varied the capacity of the SSD to test whether the advisor was able to exploit different mixtures of SSD and disk storage.

For these experiments, we used the OLAP8-63 workload. When the storage targets are heterogeneous, it is not obvious how to define baseline layouts against which to compare the layout advisor's recommendations. As our first baseline we used SEE, although it is clear that this will lead to load imbalance when the targets are heterogeneous. For the "3-1" configuration, we also considered a second baseline in which the TPC-H tables are isolated on the large target and the remaining objects are placed on the small one. For the "2-1-1" configuration, we consider a second baseline that isolates the tables on the large target, the indexes on one of the small targets, and the temporary tablespace on the other small target. These are layouts that might be considered by a database administrator faced with these situations. Finally, for the experiments involving the SSD, we also considered a baseline in which all of the TPC-H objects are placed on the SSD in those scenarios for which the SSD capacity was sufficient to permit it.

Figure 17 summarizes the results of the first experiment,

SSD Cap.	SEE	All objects on SSD	Optimized Layout	Speedup (Optimized vs. SEE)
32GB	12145	6742	6182	1.96x
10GB		n/a	6354	1.9x
6GB			6234	1.94x
4GB			8529	1.42x

Figure 18: Workload Execution Times (in second) for Baseline and Optimized Layouts on Heterogeneous Storage Targets, for the OLAP8-63 Workload

which did not involve the SSD. In addition to the results for the heterogeneous 3-1 and 2-1-1 target configurations, we have also included the results from Figure 11 for the same workload (OLAP8-63) under the homogeneous "1-1-1-1" configuration. The most important observation is that the layout advisor is able to identify a good layout regardless of the storage target configuration. In fact, the layouts it found for the heterogeneous configurations were actually slightly better than the one that it found for the homogeneous targets. Not surprisingly, the baselines do not fare as well. The performance of SEE degrades with increasing disparity in storage target configuration, faring worse under 2-1-1 than under 1-1-1-1, and worse under 3-1 than under 2-1-1. Isolating tables on the large target improved performance in the 3-1 configuration, but not as much as the optimized layout, which used 3-disk RAID0 target for the `LINEITEM` table, the `I_L_ORDERKEY`, index, and part of the `CUSTOMER`. Isolating tables and indexes in the 2-1-1 configuration hurt performance significantly relative to the SEE baseline, which points to the difficulty of using heuristic layout guidelines. In the 2-1-1 configuration, the optimized layout isolated the `LINEITEM` table on the 2-disk target and distributed the remaining objects across the 2 single-disk targets.

Figure 18 summarizes the results of the second experiment, which involved the four disk drives and the SSD. As expected, the SEE layout performed poorly because the disparity in the performance characteristics of the SSD and the disk drives. When the SSD capacity was sufficient to allow it, placing all of the objects on the SSD resulted in much better performance, but of this layout fails to utilize the disk drives at all. The optimized layout distributed the objects across the disk drives and the SSD and achieved about a 10% speedup relative to the SDD-only layout. More importantly, the layout advisor was able to determine good layouts even when the the SSD was too small to hold all of the objects. For example, even with only a 6GB SSD, the optimized lay-

Workload	N	M	Solver	Regularization	TOTAL
OLAP8-63	20	4	3.5	0.1	3.6
consolidation	40	4	12.1	0.5	12.6
		10	55	2.2	57.2
		20	120	9	129
		40	200	26	226
2xconsolidation	80	10	47	12	59
3xconsolidation	120	10	340	40	380
4xconsolidation	160	10	590	72	662

Figure 19: Execution Time of the Layout Advisor (in seconds)

out still performs better than the SSD-only layout with a 32GB SSD.

It is also instructive to compare execution times from the SSD experiments (Figure 18) to those from the disk-only experiments (Figure 17). For example, the workload runs in 16201 seconds under SEE in the four disk "1-1-1-1" configuration (with no SSD), or in 13608 seconds in the same configuration with an optimized layout (Figure 17). With the addition of a 4GB SSD to the four disks, the recommended layout allows the workload to finish in 8529 seconds (Figure 18)- almost twice as fast as the disk-only SEE layout. Thus, the layout advisor is able to exploit a small amount of added SSD capacity to achieve a substantial boost in workload performance.

6.5 Optimization Time

Figure 19 shows the time required to recommend regularized layouts, for several different workloads. For each layout we report the total time required, as well as an indication of how much of that time is spent in the NLP solver and how much is spent on the regularization step. The total time is the solver time plus the regularization time plus the time required to generate the initial configuration for the solver, which is very small (much less than a second).

The figure shows the cost for the OLAP8-63 workload and for the workload used in the consolidation scenario, which includes both the TPC-H and TPC-C database objects. In addition, we created additional synthetic workloads by taking the workload descriptions (W_i's) of the 40 objects from the consolidation workload and replicating them. This gives workloads with 80, 120 and 160 objects, labeled 2xconsolidation, 3xconsolidation, and 4xconsolidation in Figure 19. We measured the time required to generate an optimized layout for these replicated workloads on 10 storage targets.

A few things are clear from these tests. First, for layout problems at these scales (10's of targets, a few 100's of objects) the layout advisor is quite fast. For the largest problem we gave it, the total time required to generate an optimized layout was about 10 minutes. The results also show that the total optimization time is dominated by the time required by the NLP solver, rather than the regularization post-processing step. The solver timings reported in Figure 19 were obtained without any tuning of the solver. We have found that tuning can speed the solver up by a factor of two or three.

6.6 AutoAdmin Comparison

As part of the Microsoft AutoAdmin project, Agrawal et al [2] addressed a database layout problem that is similar

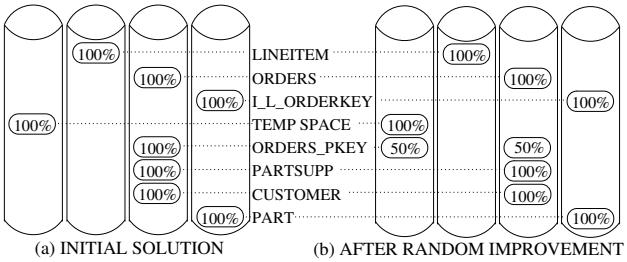

Figure 20: AutoAdmin Layout of TPC-H Database Objects for the OLAP1-63 Workload

to the one that we have considered, and they developed a tool for recommending layouts. Although that tool and our own layout advisor address similar problems, they use different approaches to solve them. Here, we present a brief comparison of the two tools that is intended to highlight the differences between their approaches.

The AutoAdmin tool takes as input a set of SQL statements describing a database system's workload. In contrast, our layout advisor expects statistical I/O workload parameters for each object. Our approach is more general in that it is not limited to layout for database systems, but for database systems the AutoAdmin input is very natural and easy to generate. The AutoAdmin tool builds a graph representation of the workload, with nodes representing objects and weighted edges between nodes representing concurrent access to those objects by workload queries. This graph is input to a two-step layout process. The first step separates heavily co-accessed objects in order to minimize interference between them. The second step further distributes objects across targets to increase I/O parallelism. In our terminology, the resulting layout is regular.

The emphasis in the AutoAdmin work is on reducing interference among concurrently access objects and on providing I/O parallelism for individual objects. It relies on relatively simple workload and performance models, e.g., it models neither workload concurrency nor performance differences among different types of storage targets. In contrast, we use more expressive models. Unlike the AutoAdmin tool, I/O parallelism for individual database objects is not an explicit objective of our layout optimization process, although our optimizer often does distribute objects across multiple targets.

Although the AutoAdmin layout technique was originally developed for Microsoft's SQLServer DBMS, we implemented it in PostgreSQL so that we could compare the layouts that it recommends with those recommended by our layout advisor. Figure 20 illustrates the layout generated by the AutoAdmin technique for our OLAP1-63 workload. Figure 20(a) shows the layout that results from the first step, in which each object is placed on a single target. Figure 20(b) shows the final layout after the second step, which attempts to distribute objects to increase the potential I/O parallelism. This final layout shares some of the features of the layout recommended by our advisor, e.g., it separates the heavily-used objects `LINEITEM`, `ORDERS`, and `I_L_ORDERKEY` from each other, and isolates the `LINEITEM` table. However, while our optimizer distributed `LINEITEM` across two storage targets, the AutoAdmin optimizer assigns `LINEITEM` to a single target, so that it is able to isolate the `TEMP SPACE` object on the remaining target. The reason for this differ-

ence is that the AutoAdmin tool relies in part on cardinality estimates from the database system's query optimizer to estimate the I/O load on objects, and these estimates are sometimes erroneous. The PostgreSQL query optimizer makes errors of multiple orders of magnitude in estimating the sizes of some intermediate objects produced by query execution plan for TPC-H Q18. This leads the AutoAdmin tool to overestimate the importance of separating `LINEITEM` and `TEMP SPACE`. Of course, this particular cardinality estimation error is an artifact of PostgreSQL and may not occur in another database system. However, most query optimizers are subject to some kinds of estimation errors.

The layout shown in Figure 20(b) is less balanced than that of Figure 1, primarily because `LINEITEM` is placed on a single target. Despite this, AutoAdmin's layout provided about the same speedup (between 1.25x and 1.3x) in workload execution time as our optimized layout did. The OLAP1-63 workload runs in 32634 seconds under AutoAdmin layout, as compared to 31789 seconds under the layout of Figure 1 and 40927 seconds under the SEE baseline layout. The imbalance of the AutoAdmin layout did not result in a significant workload execution time penalty because the storage targets, even the one holding `LINEITEM`, are lightly utilized under this workload on our test platform.

As noted previously, the AutoAdmin tool is oblivious to the concurrency level in the workload. As a result, AutoAdmin layout tool gives exactly the same layout for the OLAP8-63 workload as it does for OLAP1-63 workload because these two workloads are composed of exactly the same queries and differ only in their concurrency level. However, as discussed in Section 6.2, the workload characteristics of the TPC-H objects under OLAP1-63 and OLAP8-63 workloads are quite different. As a result, the AutoAdmin-recommended layout actually hurts performance (relative to the SEE baseline) under the OLAP8-63 workload. The OLAP8-63 workload runs in 19937 seconds on the layout shown in Figure 20(b), compared to 16201 seconds for the SEE baseline and 13608 seconds (1.19x speedup) for the layout recommended by our advisor.

For the workloads we tested, our AutoAdmin implementation produced a layout more quickly than our MINOS-based layout advisor. For example, AutoAdmin required 1.8 seconds for the OLAP8-63 workload, which is about half of what our layout advisor required. We also observed that graph partitioning (separating highly co-accessed objects) is the most time consuming step of the AutoAdmin layout algorithm.

7. RELATED WORK

File assignment problems involve assigning each of N files to one of M identical storage devices, usually with the objective of balancing the load across the devices [7, 14]. The workload models and objective functions used for file assignment problems are usually simple, e.g., each file might be associated with a numeric request rate. Issues like interference between co-located objects are not considered.

Rotem et al. [20] specifically focus on placing database objects, rather than generic files. They address the problem of laying out N base tables on M identical disk drives, with the goal of reducing the I/O cost of a given query workload. Tables may be replicated, but each replica is placed on a single disk. The query workload is assumed to be a set of 2-way join queries. The I/O cost of a query is lower if its inputs can be found on different disks than if they must be retrieved from the same disk. The optimization task is formulated as an integer programming problem. Rubio et al. [21] also consider the problem of placing database objects, given a query workload. In their case, the problem is to place the objects on the nodes of a distributed database system so that the inter-node traffic required to handle the given query workload will be minimized. The optimization task is formulated as a problem of clustering nodes in a graph which describes data traffic among nodes, and simulated annealing is used to search the solution space. The database layout tool that was developed as part of the Microsoft's AutoAdmin project [1, 2] is described in Section 6.6.

Our Rome-style approach to workload and performance modeling is based on work done at HP Laboratories. That work generated a number of tools for automating capacity planning, configuration, and other aspects of storage systems design and management [3, 6, 23], culminating in the Disk Array Designer (DAD) [5]. A recent paper [29] provides a retrospective overview of this work.

DAD automates the design of a storage system, given a description of the anticipated workload. One of the key problems addressed by DAD is capacity planning. In other words, the number of available storage targets is not an input to DAD. Instead, DAD attempts to determine how many targets (and what types of targets) are needed to support the given workload. In addition, DAD determines where to place individual objects with the storage system that it designs, which is essentially the layout problem that we consider. Thus, layout is one part of a broader optimization problem than the one that we have considered. However, DAD only generates layouts in which each object is assigned to a single target. To explore its space of potential system configurations and layouts, DAD uses an ad hoc technique involving an initial bin-backing step followed by randomized search. Moves in this search space include steps like adding additional capacity to the current configuration or assigning a particular object to a storage target. It should be possible to design a similar randomized search technique to solve the layout problem faced by our layout advisor - this would be an alternative to the NLP solver that we used.

8. CONCLUSIONS AND FUTURE WORK

We have presented a technique for recommending optimized layouts of database objects onto storage targets, such as disk drives, SSDs, or RAID groups. The technique leverages input workload descriptions and storage target models to avoid potential interference among co-located objects, and to ensure that the recommended layout is balanced. It also uses the storage target models to ensure that the recommended layout reflects the distinct performance characteristics of each target.

Our technique uses a generic NLP solver along with several heuristics that are specific to the layout problem. The technique could be deployed as a standalone storage layout advisor, whose output would guide the configuration of both the database system and the storage system. We demonstrate empirically that such a layout advisor can quickly recommend effective layouts over both homogeneous and heterogeneous storage target configurations.

A layout advisor that implements our proposed layout technique can provide layout recommendations to a database administrator or storage administrator. This allows the ad-

ministrators to define logical storage volumes, containers, or other constructs with which to implement the recommended layout. However, it should also be possible to utilize layout recommendations in situations in which data placements decisions are made more dynamically. For example, NetApp storage systems [17] employ a feature called flexible volumes, or FlexVols [9]. FlexVols use a shared pool of storage resources, and the capacity of a FlexVol grows dynamically and only when new data is actually written to the flexible volume. Thus, instead of statically assigning disks and fixed capacity to volumes during an initial configuration step, capacity is assigned dynamically as the system runs. The current FlexVol implementation results in a SEE layout of the volume over the underlying storage targets, but the layout techniques described in this paper could be used to guide the storage system's dynamic allocation decisions as FlexVols grow.

Finally, it would be useful to extend the layout advisor so that it recommends storage configurations in addition to layouts. Instead of taking a set of storage targets as input, the advisor would instead take a description of the available unconfigured storage resources. The advisor's output would recommend how to configure specific storage targets, e.g., RAID groups, from the available resources, as well as how to lay out objects onto the targets. This would move the layout advisor a step in the direction of tools such as Minerva and DAD, which use heuristics to attack an even broader problem.

9. ACKNOWLEDGEMENTS

This work was supported by NetApp and the Natural Sciences and Engineering Research Council of Canada.

10. REFERENCES

[1] G. Aggarwal, T. Feder, R. Motwani, R. Panigrahy, and A. Zhu. Algorithms for the database layout problem. In *Proc. Int'l Conf. on Database Theory (ICDT)*, pages 189–203, 2005.

[2] R. Agrawal, S. Chaudhuri, A. Das, and V. R. Narasayya. Automating layout of relational databases. In *Proc. IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 607–618, March 2003.

[3] 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 Transactions on Computer Systems*, 19:483–518, 2001.

[4] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4, HP Labs, 2001.

[5] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Ergastulum: Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, 2005.

[6] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *Proc. of the 5th IFIP Workshop on QoS*, pages 199–202, 1997.

[7] H.-J. Chen and T. D. C. Little. Physical storage organizations for time-dependent multimedia data. In *Proc. Conf. on Foundations of Data Organization and Algorithms (FODO)*, pages 19–34, 1993.

[8] Transaction Processing Performance Council. TPC-C and TPC-H Benchmarks. http://www.tpc.org/

[9] J. K. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. A. Smith, and E. Zayas. Flexvol: flexible, efficient file volume virtualization in WAFL. In *Proc. USENIX Annual Tech. Conf.*, pages 129–142, 2008.

[10] EMC Corp. EMC symmetrix DMX-4 series, Feb 2009.

[11] R. Fourer, D. M. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Thomson Brooks/Cole, 2nd edition, 2003.

[12] Hitachi Data Systems Corp. Hitachi universal storage platform V, October 2008. http://www.hds.com/assets/pdf/hitachi-universal-storage-platform-family-architecture-guide.pdf

[13] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In J. T.-L. Wang, editor, *Proc. ACM Int'l Conf. on Management of Data (SIGMOD)*, pages 1075–1086. ACM, 2008.

[14] Y.-C. Ma, J.-C. Chiu, T.-F. Chen, and C.-P. Chung. Variable-size data item placement for load and storage balancing. *Journal of Systems and Software*, 66(2):157–166, 2003.

[15] M. Mesnier, M. Wachs, B. Salmon, and G. Ganger. Relative fitness models for storage. *SIGMETRICS Performance Evaluation Review*, 33(4), 2006.

[16] B. Murtagh and M. Saunders. Minos: A projected lagrangian algorithm and its implementation for sparse nonlinear constraints. *Mathematical Programming Study*, 16:84–117, 1982.

[17] NetApp Inc. Fabric attached storage, February 2009. http://www.netapp.com/us/products/storage-systems/fas6000/

[18] Oracle. Take the guesswork out of database layout and I/O tuning with automatic storage management. Oracle Technical White Paper, December 2005.

[19] O. Ozmen, K. Salem, M. Uysal, and M. H. S. Attar. Storage workload estimation for database management systems. In *Proc. ACM Int'l Conf. on Management of Data (SIGMOD)*, pages 377–388, 2007.

[20] D. Rotem, G. A. Schloss, and A. Segev. Data allocation for multi-disk databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):882–887, 1993.

[21] J. Rubio, C. Lefurgy, and L. K. John. Improving server performance on transaction processing workloads by enhanced data placement. In *Proc. IEEE Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 84–91, 2004.

[22] A. Sachedina, M. Huras, and A. Colangelo. Best practices database storage. White paper, IBM DB2 for Linux, UNIX, and Windows, Oct. 2008.

[23] E. Shriver. A formalization of the attribute mapping problem. Technical Report HPL-SSP-95-10, HP Labs, 1996.

[24] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecom. Systems (MASCOTS)*, pages 183–192. IEEE, Aug. 2001.

[25] E. Varki, A. Merchant, J. Xu, and X. Qiu. An integrated performance model of disk arrays. In *Proc. Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecom. Systems (MASCOTS)*, pages 296–305, 2003.

[26] A. Veitch and K. Keeton. The rubicon workload characterization tool. Technical Report HPL-SSP-2003-13, HP Labs, March 2003.

[27] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger. Storage device performance prediction with CART models. In *Proc. ACM SIGMETRICS*, pages 412–413, 2004.

[28] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proc. Int'l Workshop on Quality of Service (IWQoS)*, pages 75–91, 2001.

[29] J. Wilkes. Traveling to Rome: a retrospective on the journey. *SIGOPS Oper. Syst. Rev.*, 43(1):10–15, 2009.