Automated Storage Layout for Database Systems

by

Oguzhan Ozmen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modern storage systems are complex. Simple direct-attached storage devices are giving way to storage systems that are flexible, network-attached, consolidated and virtualized. Today, storage systems have their own administrators, who use specialized tools and expertise to configure and manage storage resources. As a result, database administrators are no longer in direct control of the design and configuration of their database systems' underlying storage resources. This introduces problems because database physical design and storage configuration are closely related tasks, and the separation makes it more difficult to achieve a good end-to-end design. For instance, the performance of a database system depends strongly on the storage layout of database objects, such as tables and indexes, and the separation makes it hard to design a storage layout that is tuned to the I/O workload generated by the database system. In this thesis we address this problem and attempt to close the information gap between database and storage tiers by addressing the problem of predicting the storage (I/O) workload that will be generated by a database management system. Specifically, we show how to translate a database workload description, together with a database physical design, into a characterization of the I/O workload that will result. Such a characterization can directly be used by a storage configuration tool and thus enables effective end-to-end design and configuration spanning both the database and storage tiers. We then introduce our storage layout optimization tool, which leverages such workload characterizations to generate an optimized layout for a given set of database objects. We formulate the layout problem as a non-linear programming (NLP) problem and use the I/O characterization as input to an NLP solver. We have incorporated our I/O estimation technique into the PostgreSQL database management system and our layout optimization technique into a database layout advisor. We present an empirical assessment of the cost of both tools as well as the efficacy and accuracy of their results.

Acknowledgements

I would like to thank my supervisor Kenneth Salem for his invaluable guidance and support throughout my PhD studies. His continuous encouragement and positive attitude made it not only possible to earn my PhD degree but also a delightful experience. Among many lessons I learned from him, one has become a pillar in my professional life: think simple!

I also would like to thank my thesis committee at the University of Waterloo, Tamer Ozsu, Ihab Ilyas, Patrick Lam and Mahesh Tripunitara, and my external examiner Amr El Abbadi from University of California at Santa Barbara for reviewing my thesis.

I am grateful to both HP Labs Storage Group and Network Appliance for their support to our projects. This thesis has been greatly inspired by the I/O workload characterization and capacity planning projects held at HP Labs. I would like to thank HP Labs for sharing their code base and Mustafa Uysal, a member of Storage Group, for his involvement in our I/O workload estimation project. I am indebted to Network Appliance for their financial support to our storage layout project. In particular, I would like to thank Jiri Schindler from NetApp for his direct involvement in this project.

Last but not least, I'd like to thank my mother, my father and the rest of my family for their support and pray.

Contents

List of Figures

Chapter 1

Introduction

The complexity of modern enterprise computing environments is prompting changes in the way that computing resources and the systems that depend on them are deployed and managed [22, 28, 49, 50, 58]. As a major component of a computing environment, storage systems are getting more sophisticated, too. Simple, direct-attached storage devices are giving way to flexible, network-attached, consolidated and virtualized storage systems. Increasingly, storage resources are consolidated into a common pool, virtualized to accommodate individual application requirements, and shared by multiple enterprise applications, including database management systems (DBMS). Furthermore, in order to effectively address this complexity, storage resources are increasingly administered separately from the server infrastructure. Storage administrators (SAs) are expected to balance the requirements of multiple database systems and other storage clients by applying their own domain knowledge and expertise. As a result, database administrators (DBAs) are no longer in direct control of the design and configuration of their database systems' underlying storage resources.

Managing the storage infrastructure is, like database administration, a complex task. A storage administrator has to configure storage arrays, create logical units at storage arrays, create logical volumes at servers, configure storage controllers and storage network switches with appropriate access credentials, and manage the ongoing usage of the storage devices to prevent performance bottlenecks or resource shortages. The configuration decisions made by the SA determine the performance, reliability, and capacity characteristics of the storage system as seen by the DBMS. Good configuration decisions should take into account the characteristics of the storage (I/O) workload generated by the DBMS.

To help SAs cope with the complexity of these storage configuration tasks, researchers have developed tools that can be used to automate them [7, 8, 27, 56]. Those tools rely on storage workload information in some form. Hence, effective storage administration, whether manual or automatic, depends on the knowledge of *storage system workload*. Although storage workloads can be described in different ways, e.g., as a sequence of I/O requests or as a set of statistical parameters, their

purpose is to describe the I/O characteristics or behavior of the storage clients. In general, a storage workload description identifies the I/O access characteristics for each of the distinctly addressable chunks of data, such as files or logical volumes, accessed by the storage clients.

Accurate workload characterizations can be difficult to obtain, particularly at initial configuration time. Before the system is operational and it is not possible to observe the storage clients' I/O behavior, a storage administrator can depend only on generic guidelines and rough "guesstimates". Once the storage system is operational, workload characteristics can be observed. However, such observations are not a panacea: they may be expensive to obtain and use, they do not solve the initial configuration problems, and they are of no use in addressing "what if" questions. For example, a DBA may be considering a possible physical design change such as the creation of an index. If created, this index would affect the I/O workload experienced by the underlying storage system. Direct observation of the current storage system workload does not by itself provide any guidance as to what the storage workload would look like if the index were added.

In the first part of the thesis, specifically in Chapter 2, we attempt to close the information gap between the database tier and the storage tier by addressing the problem of predicting the storage (I/O) workload that will be generated by a database management system. By estimating database systems' storage workloads, we can provide storage administrators with information that they can use to make informed planning, design, and configuration decisions. Storage workload estimation provides an SA with not only a faster alternative to observing an operational system and collecting the actual workload but also a means to obtain a storage workload description at initial configuration time. In addition, workload estimation makes it possible to obtain the storage workload for a hypothetical database physical design. For example, the effect of adding an index on the storage workload can be estimated without materializing the index.

By providing storage workload estimations, we also enable end-to-end solutions to database physical design and storage configuration problems. With storage workload estimation, both the DBA and SA have sufficient information to address their part of the end-to-end design and configuration problem. One example of this is shown in Figure 1.1, which illustrates how existing database physical design tools and storage configuration tools could be combined to determine both a database physical design and an appropriate storage configuration for a given database workload, while preserving the administrative autonomy of the database and storage tiers.

In the second part of the thesis, specifically in Chapter 3, we show how such storage workload information can be leveraged to improve and facilitate one of the aspects of storage system design. Specifically, we focus on storage layout design for database management systems. Storage layout is an important part of storage configuration. In brief, a layout describes how logical volumes which contain storage clients' data are mapped to actual physical storage devices. A DBMS relies on an

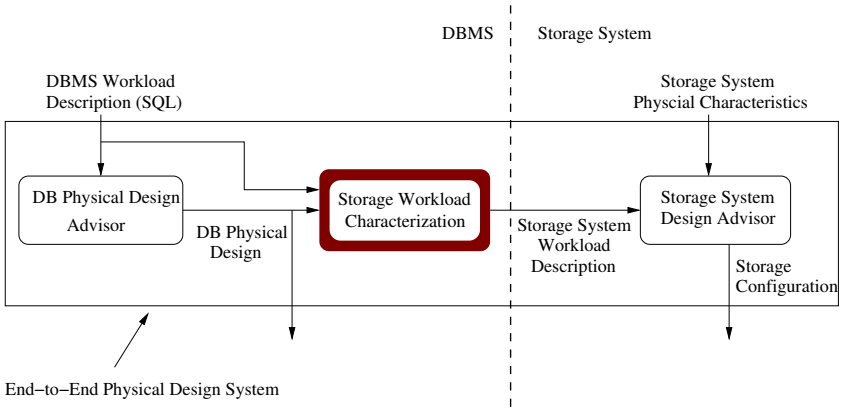


Figure 1.1: End-to-End Physical Design Using Existing Design Advisors

underlying storage system for persistent storage of database objects such as tables, indexes, and logs, and the performance of the database system depends strongly on the layout of those objects. In Chapter 3, we describe our technique to find a good layout for a given storage workload; that is, a storage layout which is tuned to the given storage workload. A good layout will both balance the storage workload generated by the database system and avoid performance-degrading interference that can occur when concurrently accessed objects are stored on the same volume.

In summary, this thesis addresses two complementary and independent problems. First, we address the problem of predicting the storage workload that will be generated by a database management system. Second, we focus on one aspect of storage system design, and describe a technique which leverages I/O workload information to recommend a storage layout for database objects.

1.1 Contributions

This thesis makes the following contributions:

- We formulate the storage workload estimation problem for relational database management systems. In our formulation, storage workloads are described in a domain-independent and configuration-independent language called Rome [60]. By "domain-independent", we mean that the workload description that is produced is not specific to database management systems. Similar descriptions can be produced for other storage system clients. As storage consolidation becomes more common, this property becomes more important. Using a common and generic storage workload model makes it feasible to aggregate workload descriptions from multiple storage applications, including database management systems.

- We present a technique for producing storage workload estimates. Those estimates not only are a faster alternative to monitoring a running system to obtain I/O workload information but they also make it possible to generate I/O workload information at initial configuration time and for hypothetical database physical designs. Our technique has been implemented in the context of the PostgreSQL DBMS.

- We present an empirical evaluation of the accuracy of the storage workload estimates produced by our technique, and the cost of producing them.

- 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 under various scenarios. We also compare our methodology with a recently proposed related work.

1.2 Organization of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 explains our methodology for estimating the I/O workload that is generated by a database management system. Chapter 3 describes the layout optimization technique. Both Chapter 2 and Chapter 3 present experimental evaluations of the proposed methods. Chapter 4 discusses the existing work related to I/O workload estimation and data layout. Finally, Chapter 5 concludes the thesis.

Chapter 2

Storage Workload Estimation for Database Systems

Modern storage systems are complex. As was stated in the preceding chapter, storage resources are increasingly administered separately from the server infrastructure to effectively address this complexity. Therefore, database administrators (DBAs) are no longer in direct control of the design and configuration of their database systems' underlying storage resources. Storage administrators (SAs) are expected to balance the requirements of multiple database systems and other storage clients. Configuration decisions made by the SA determine the performance, reliability, and capacity characteristics of the storage system as seen by the database management system (DBMS). As a result, the separation of the management of database systems and storage systems may introduce problems if storage configuration decisions do not take into account the characteristics of the storage (I/O) workload generated by the DBMS. Effective storage administration depends on knowledge of the storage system workload.

A storage workload description provides an SA with valuable information on the I/O characteristics or behaviour of the storage clients. Although storage workloads can be described in different ways, such as a sequence of block I/O requests or a set of statistical parameters, each description identifies the I/O access characteristics for one of the distinctly addressable chunks of data, such as files or logical volumes accessed by the storage clients. Provided with database systems' storage workloads, SAs can make informed planning, design, and configuration decisions. In fact, to help SAs cope with the complexity of storage configuration, researchers have developed tools to automate storage configuration tasks [7, 8, 27, 56]. Those tools rely on storage workload information in some form.

In this chapter we address the problem of predicting the storage workload that will be generated by a database management system. Specifically, we show how to translate a database workload description, together with a database physical design, into a characterization of the storage workload that will result. Our I/O workload estimation technique produces storage workloads described in a domain-

independent and configuration-independent language called Rome [60]. By "domain-independent", we mean that the workload description that is produced is not specific to database management systems. Similar descriptions can be produced for other storage system clients. Since shared, consolidated storage systems must accommodate workloads from a variety of clients, including databases, it is important to target a generic workload model. Doing so allows an SA to aggregate workload descriptions from multiple storage applications. As storage consolidation becomes more common, this property becomes more important.

The remainder of this chapter is structured as follows. Section 2.1 describes the storage workload estimation problem and defines the target storage workload model. Section 2.2 presents the proposed workload estimation technique and its implementation in PostgreSQL DBMS, and Section 2.3 describes its evaluation. Finally, Section 2.4 concludes the chapter.

2.1 Storage Workload Estimation Problem

In this section, we will define the problem of estimating storage workload characteristics given a specification of the database workload. To formulate this problem more precisely, "database workload" and "storage workload" must be defined first.

2.1.1 Database Workload Model

Existing relational database design tools typically expect the database workload to be defined as a set of SQL statements along with some indication of the relative frequency of occurrence of each statement [3, 62]. We use a similar characterization of the database workload for the storage workload estimation problem, so that a single workload description can be used for both tasks. Specifically, it is assumed that the workload is characterized by a fixed set $\mathcal{Q}$ of SQL statements defined over a known database schema. We refer to each such statement as a *query type*. Each query type $\mathcal{Q}_i$ has an associated weight f_i which represents its prevalence in the workload. The proportion of queries of type $\mathcal{Q}_i$ in the workload is given by $\frac{f_i}{\sum_i f_i}$.

This kind of database workload characterization describes the mix of queries in the database workload. This is sufficient for tasks such as index selection, where the goal is to choose a set of indexes that will provide superior performance relative to the performance achievable using other sets of indexes. However, we would like our storage workload estimates to be useful for a variety of storage management tasks, including those that require information about *absolute* frequency of occurrence of the various queries. An example of such a task is capacity planning. To enable this, it is also required that the database workload description include a specification of a *target operating point* for the database system. We use two parameters to characterize an operating point. The first is the total query throughput, denoted

Symbol	Description
$\mathcal{Q}$	set of possible SQL statements (query types)
f_i	relative frequency of query type $\mathcal{Q}_i$
$\mathcal{T}$	query throughput (in queries per second)
CL	number of concurrent queries
$\mathcal{D}$	set of database physical objects

Figure 2.1: Database Workload Model Parameters

by $\mathcal{T}$. The second is the query multiprogramming or concurrency level, CL, which describes the expected number of concurrently executing queries at any given time.

Finally, since the proposed storage workload estimator relies on the database system's query optimizer, it is required that optimizer be configured to behave as it would at the target operating point. In particular, database statistics should be available so that the query optimizer will choose appropriate query execution plans. Again, existing database administration tools have similar requirements for the availability of statistics, and some database systems support the definition of hypothetical database instances to support cost-based "what if" analyses without the need to populate the hypothetical instance [13].

It is assumed that a database physical design has been selected, perhaps through the use of a physical design advisor [3, 62], and that the physical design is known to the query optimizer. We use $\mathcal{D}$ to represent the set of physical database objects: tables, indexes, tablespaces and so on. Figure 2.1 summarizes the database workload parameters.

2.1.2 Storage Workload Model

One way to characterize I/O workloads is to use a trace of I/O events, or a set of traces. Although traces are a very detailed and expressive way to describe storage workloads, they have some disadvantages. They are large and expensive to store and manipulate. Traces of database I/O workloads are also expensive to collect, as collection requires populating the database and applying a realistic load. Trace-based workload descriptions cannot be used as input to analytical models of storage system behavior. Finally, traces tend to be specific to a particular storage configuration, and difficult to generalize. It is prohibitively expensive to collect traces from multiple candidate storage configurations.

Instead, we adopt a more abstract I/O workload model called the *Rome* model [60, 61]. The Rome model is the unifying "glue" for a collection of storage management tools that support performance modeling, capacity planning, storage system design and configuration, and other tasks [7, 8, 56]. The Rome model is *not* specifically designed to model the I/O workloads generated by database management systems. It is a general purpose model intended to model storage workloads generated by

Symbol	Description
t_{on}	burst duration (in seconds)
t_{off}	inter-burst gap (in seconds)
λ^r	read request rate during bursts (in requests per second)
λ^w	write request rate during bursts (in requests per second)
B^r	average size of read requests (in bytes)
B^w	average size of write request (in bytes)
Q	average block count of sequential runs
$\Phi_i[j]$	burst overlap between streams i and j

Figure 2.2: I/O Request Stream Parameters in the Rome Model

any kind of storage client. Since shared, consolidated storage systems must accommodate workloads from a variety of clients, including databases, it is important to target a generic workload model. Doing so allows a storage administrator to aggregate workload descriptions from multiple storage applications. By targeting the Rome model in particular, it is also possible to leverage existing Rome-based workload analysis and storage management tools.

The Rome model views the storage system abstractly, as a set of *stores*. A store can be thought of as a virtual block storage device, disjoint from other stores, to which block read and write requests can be directed. The I/O workload directed to a store is represented by one or more concurrent *streams*. A stream consists of bursts of I/O request activity of duration t_{on} interleaved with idle periods of duration t_{off}, during which no requests occur. During each on burst, read requests to the underlying store occur at rate λ^r and write requests occur at rate λ^w.

Each I/O request has a starting position (within the underlying store) and a size, or length, B^r for read requests and B^w for write requests. The starting position of each request is determined by a *run count* parameter Q. Successive requests in a stream start where the previous request left off, until the total number of requests in the run reaches Q. The next request then starts a new run, with a randomly chosen starting position. Thus, $Q = 1$ models a random I/O request pattern, while larger values of Q model sequentiality. Figure 2.2 summarizes the parameters associated with a Rome request stream. Together, these parameters describe the request stream properties that are important to the underlying storage modeling and management tools: request rates, read/write mix, burstiness, request size, and sequentiality.

In addition to these per-stream properties, Rome also describes burst correlations, which model the amount of temporal overlap among the bursts of different streams. Given a set S of streams, Rome defines an $|S| \times |S|$ *overlap matrix* Φ. Entry $\Phi_i[j]$ in the overlap matrix, $1 \leq i, j \leq |S|$, describes the percentage of stream i's burst period during which stream j is also active. Note that, as defined by the Rome model, the overlap matrix need not be symmetric. For example, consider two streams S_i and S_j, with $t_{on}[i] = 100$ and $t_{on}[j] = 10$, for which S_j's bursts are

completely contained within S_i's bursts. This will be described by $\Phi_i[j] = 10\%$ and $\Phi_j[i] = 100\%$.

2.1.3 Problem Statement

With the definitions of a database workload and storage workload in place, the problem can be stated as follows:

Definition 1 ***Storage Workload Estimation Problem:*** *Given a database workload characterization, including a target operating point, and a database physical design, produce an I/O workload characterization that accurately models the storage workload that will be generated by the database system under the given database load at the target operating point.*

In general, to estimate a Rome storage workload characterization, it is necessary to address several questions:

- How many stores should the model have?
- How many request streams should each store have?
- What stream parameter settings should be used for each stream?

Here, the workload estimation problem is simplified by fixing the answers to two of these questions, thus restricting the space of workload models that can potentially be generated by the estimator. First, only workload models that include exactly $|\mathcal{D}|$ Rome stores (one for each physical database object) are considered. There is little reason to have more than one store per physical database object, since this provides sufficiently fine granularity in the workload description for most storage configuration tasks. Second, only workload models with a single request stream per store are considered. A natural alternative to this would allow up to $|\mathcal{Q}|$ request streams for each store, where each stream would describe the I/O requests generated by queries of a particular type against a particular physical database object. In contrast, single-stream-per-store models must use a single set of stream parameter settings to characterize the aggregate workload of all types of queries against a given store. We focus on single-stream-per-store models here because they are simpler. However, the storage estimation method described in the following section can easily be extended to generate $|\mathcal{Q}|$-streams-per-store if additional expressiveness is required. Furthermore, existing Rome-based storage management tools can accommodate multi-stream stores.

1: **for** $\mathcal{Q}_i$ in $\mathcal{Q}$ **do**
2: Use the database query optimizer to obtain a plan for $\mathcal{Q}_i$
3: Generate an I/O request sequence $\mathcal{RS}_i$ for $\mathcal{Q}_i$'s plan.
4: **end for**
5: Merge the $\mathcal{RS}_i$ to produce a representative I/O request trace `Tr`
6: **for** $\mathcal{D}_j$ in $\mathcal{D}$ **do**
7: Extract the representative request trace $\texttt{Tr}_j$ for $\mathcal{D}_j$ from `Tr`
8: Fit Rome model parameters to $\texttt{Tr}_j$
9: **end for**

Figure 2.3: Storage (I/O) Workload Estimation Algorithm

2.2 Estimating Storage Workload

Figure 2.3 gives a high-level outline of the proposed method of estimating a Rome I/O workload model. As described in Section 2.1.3, the output of this method is one set of Rome I/O model parameter values (as shown in Figure 2.2) for each physical database object $\mathcal{D}_j \in \mathcal{D}$. The model parameters for $\mathcal{D}_j$ describe the I/O workload that the DBMS is expected to apply to the stored representation of that object.

The method shown in Figure 2.3 consists of three phases. First, the estimator generates an I/O request sequence corresponding to each query type in the database workload in isolation (Figure 2.3 lines 1-4). Second, it merges those individual sequences into a single I/O request trace, which is called the *representative I/O trace* for the given database workload and operating point (line 5). Finally, it projects each physical object's requests from the representative trace and fits the Rome stream parameters to the projected trace (lines 6-9). What follows is a detailed description of these phases.

2.2.1 Estimating Query I/O Request Sequences

An *I/O request sequence* is an ordered list of records, each of which describes a single block I/O operation. Specifically, each record consists of the following fields: physical object identifier, starting offset within the physical object, request size, and request type (read or write). Note that, in Figure 2.3, request sequences ($\mathcal{RS}_i$'s) have been distinguished from *request traces* (`Tr` and $\texttt{Tr}_j$'s). A request trace differs from a request sequence in that the former includes timing information for each I/O operation, while the latter does not.

The first phase of the storage workload estimation process is to predict a separate I/O request sequence for each type of query in the database workload. These request sequences describe the I/O behavior of a single query running *in isolation*. Figure 2.4 summarizes the approach.

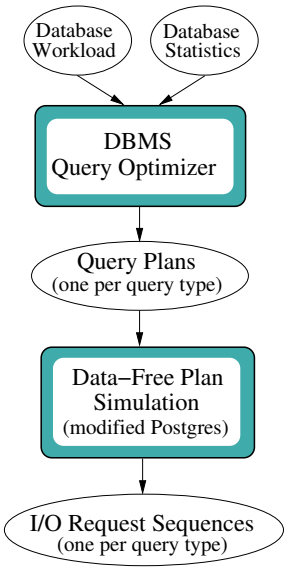


Figure 2.4: Generating Query I/O Request Sequences

To obtain these sequences, we perform a *data-free simulation* of the control flow of each query's execution plan. A query execution plan is a tree in which the nodes represent database query execution plan operators such as table scan and merge join and the edges represent the data flow between operators. During the data-free simulation of a plan, the plan operators generate I/O records describing any I/O operations that they would have generated during a normal plan execution. However, they do *not* actually generate the I/O operations. Instead, they log the descriptions of those operations as if they have been actually issued. These I/O records are concatenated to form the I/O request sequence for the query.

When a query plan is actually executed by the database system, its control flow depends on the data that is flowing through the plan. During the data-free simulation, operators neither retrieve the data nor flow the data through the plan. The simulation relies instead on the cardinality estimates produced by the query optimizer to approximate the control flow that would have occurred during an actual execution of the plan. For example, for a tuple-oriented nested loop join, we use the optimizer's estimate of the cardinalities of the inner and outer relations and its estimate of the join selectivity to estimate the number of times that the join operator's left and right children in the plan will be asked to produce data. The simulation also relies on some operator-specific assumptions. For example, a (external) sort operation is assumed to create initial runs that are twice the size of the working memory available for the sort operator.

By performing the data-free simulations, we attempt to capture several important properties of the I/O workload that will be generated by queries of each type. First, the resulting I/O sequences will contain the correct numbers of I/O requests for each physical database object used by the query, up to the accuracy

of the query optimizer's cardinality estimates and the simplifying assumptions in the simulation. Second, the I/O request sequences will distinguish sequential and random I/O, based on the type of operator that is generating the requests and on information from the database catalog. For example, a table scan of a relation will generate sequential requests, while an index scan of the same relation using an uncorrelated (i.e., unclustered) secondary index will generate random requests. Finally, the sequence will capture the interleaving of requests for the various physical database objects used by the query plan. For example, the simulation understands that a hash join will first retrieve the entire build input and then retrieve the entire probe input, resulting in non-interleaved access to the physical objects that provide the build and probe inputs. Conversely, a nested loop join will result in interleaved accesses to the inner and outer inputs.

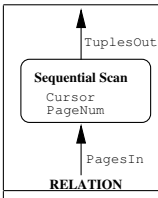
Implementation of data-free simulation is embodied in a modified version of PostgreSQL. In the modified version of PostgreSQL, there are 18 different operators that may appear in execution plans. The plan simulator handles most aspects of these operator types. One limitation of the current implementation is that queries referring to view definitions are not handled. This is a restriction of the current prototype, not a fundamental or technical restriction. Figure 2.5 illustrates the simulation for three of the PostgreSQL operators: table (sequential) scan, index scan, and nested loop join. Illustrations of the simulation of all of the PostgreSQL operators can be found in Appendix A. In these illustrations "ReadPage()" and "WritePage()" functions cause an I/O request to be *logged* in an I/O request sequence. Both functions accept two arguments representing an I/O request: a unique object identifier and an offset within the object. Note that request size is implicit because PostgreSQL uses an 8KB page size. There are only seven operators that can issue an I/O request. Three of them are scan operators that can be found only at the leaf level of an execution plan: table scan, index scan and tid scan operators. Other four operators can only be found in the inner nodes of a plan, and write into and read from temporary files: sort, hash, hash join and materialize operators.

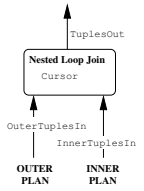
Note that data-free simulation of a query plan is generally much faster than the actual execution of the plan. This is because the simulation does not retrieve any stored data, does not flow these data through the plan operators, and does not generate any intermediate or final query results. More information about the cost of data-free simulation is given in Section 2.3.4.

2.2.2 Generating the Representative Trace

The I/O request sequences generated in the first phase capture the I/O workload characteristics of a single query running in isolation. In the second phase, the estimator generates a representative I/O trace that describes the aggregate storage workload of the entire database workload.

The generation of the representative I/O trace adds three kinds of information to the individual query request sequences. First, since the representative I/O trace

Operator	Handling Init()	Handling getNext()
	Cursor := 0; PageNum := 0;	position := ceil(Cursor); Cursor += PagesIn/TuplesOut; for i := 1 to (ceil(Cursor)-position) ReadPage(RELATION,PageNum); PageNum += 1;

	Cursor := 0;	position := ceil(Cursor);
		Cursor += OuterTuplesIn/TuplesOut;
		for i := 1 to (ceil(Cursor)-position)
		getNext(OUTERPLAN);
		Init(INNERPLAN);
		for j := 1 to InnerTuplesIn
		getNext(INNERPLAN);

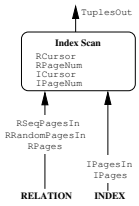
	RCursor := 0;
	RPageNum :=
	random(0,RPages-RSeqPagesIn);
	ICursor :=
	random(0,IPages-IPagesIn);
	IPageNum := ICursor;
		Iposition := ceil(ICursor);
		ICursor += IPagesIn/TuplesOut;
		Rposition := ceil(RCursor);
		RCursor += $\frac{RSeqPagesIn+RRandomPagesIn}{TuplesOut}$;
		for i := 1 to (ceil(ICursor)-Iposition)
		ReadPage(INDEX,IPageNum);
		IPageNum += 1;
		for i := 1 to (ceil(RCursor) - Rposition)
		if Rposition < RSeqPagesIn
		ReadPage(RELATION,RPageNum);
		RPageNum += 1;
		Rposition += 1;
		else
		pagenum := random in [0,..,RPages];
		ReadPage(RELATION,pagenum);

Figure 2.5: Data-Free Simulation of the PostgreSQL Plan Operators. In the diagram, operators are annotated with the names of state variables maintained by the simulation. Operator inputs and outputs are annotated with the names of PostgreSQL optimizer statistics and configuration parameters that are used by the simulator.

describes the aggregate storage workload generated by the database system, it reflects the mixture and frequency of the various types of queries that make up the database workload. Second, it accounts for the effect of buffer caching and prefetching on the aggregate I/O stream. Finally, unlike the per-query request sequences, the representative trace incorporates timing information in the form of an arrival timestamp for each I/O request. These timestamps reflect the I/O request throughput that will be required to support the database system at the specified operating point.

Figure 2.6 summarizes the process of generating the representative I/O trace. A simple probabilistic operational model of the database system is used to generate a merged I/O sequence from the per-query I/O sequences obtained in the first phase. The database system is assumed to have a fixed query multiprogramming level `CL` at

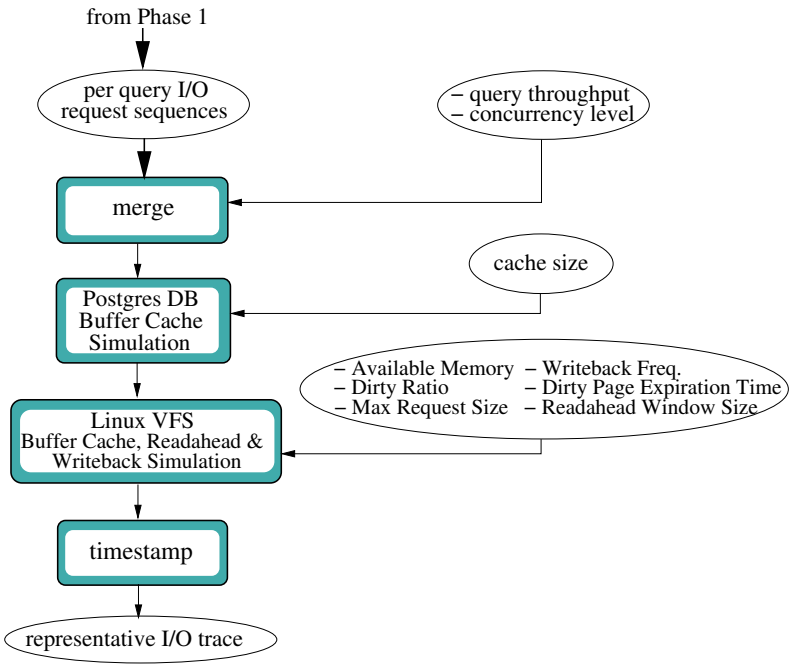


Figure 2.6: Generating the Representative I/O Trace

the target operating point. `CL` is specified as a workload parameter (see Figure 2.1). To generate a merged I/O sequence, `CL` query types are selected at random, with query type i selected with probability proportional to f_i. The I/O sequences for the selected query types are then round-robin merged to produce a single request sequence. When one of the per-query sequences is exhausted during the merger, another query type is selected and its I/O sequence replaces the exhausted one. This generative process continues until a specified number of per-query I/O sequences have been merged.

As the merged request sequence is formed, the estimator passes it to a DBMS-specific buffer cache model. To model the buffer cache, the estimator is currently employing a simulation of the 2Q cache replacement algorithm [26] that is used by PostgreSQL version 8.0.6. This simulation is parametrized by the buffer cache size. The effect of the simulation is to remove from the request sequence any I/O requests that hit the (simulated) buffer cache. Note that the buffer cache simulator does not store any actual data. It merely stores information (physical object identifier and offset within the object) which uniquely identifies each 8KB page that would be cached.

Unlike many commercial database management systems, PostgreSQL relies on the underlying operating system to carry out prefetching, and thus issues buffered I/O. This means that the data pages are also buffered in the operating system's buffer cache, and write I/O requests are deferred for some amount of time before they are submitted to the storage subsystem. Therefore, the output of the PostgreSQL buffer simulation is fed to a simulation of the Linux virtual filesystem

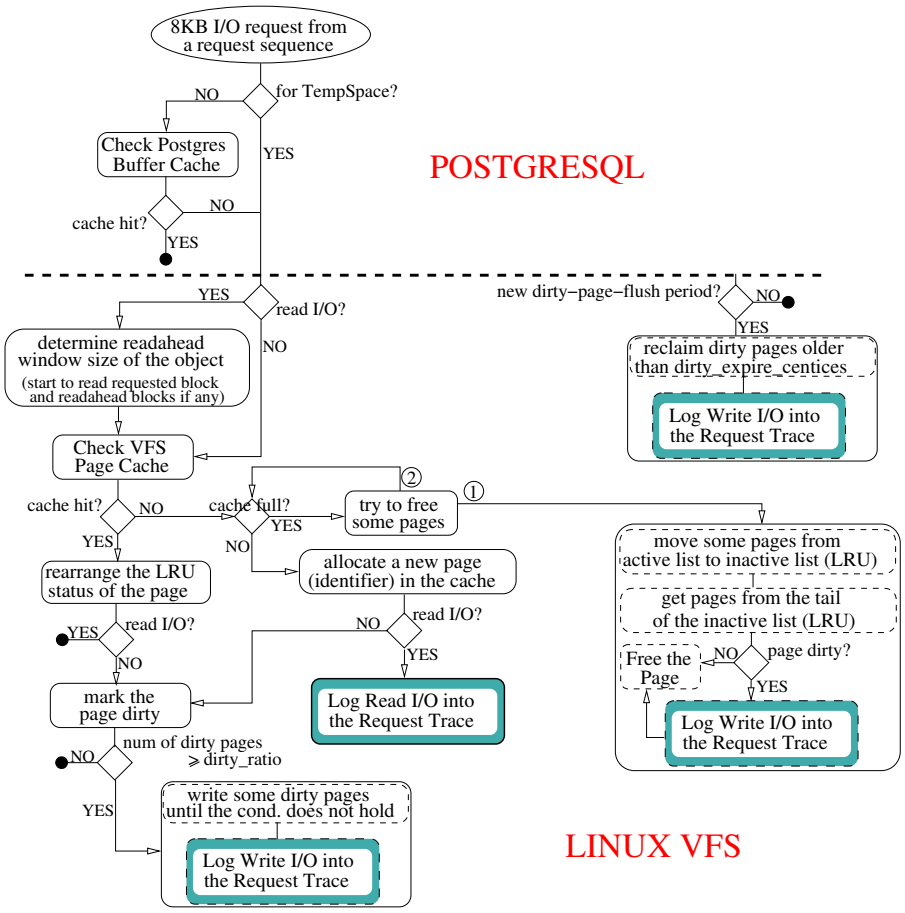


Figure 2.7: Decision Flow Diagram for PostgreSQL I/O Requests

(VFS) buffer cache. The Linux VFS buffer cache simulation is parametrized by the amount of available memory. It is assumed that the DBMS is the only application running in the system, and all of the system memory is available for data buffering except for the memory allocated to the DBMS for caching.

The current implementation uses a simulation of the two-list LRU cache replacement algorithm [10]. Writeback and readahead mechanisms implemented by the VFS are also simulated. Writeback simulation is parametrized by three parameters: the threshold that sets the maximum number of dirty pages in the cache at any time, the expiration time of dirty pages, and the period of the dirty page reclaiming process. Those parameters are tunables in a Linux system, and can be found in `/proc/sys/vm`: `dirty_ratio`, `dirty_expire_centisecs`, and `dirty_writeback_centisecs`. Readahead simulation is parametrized by two parameters: maximum readahead window size and the maximum size of an I/O request. Those parameters are tunables for each I/O request queue, and are named `read_ahead_kb` and `max_sector_kb`. Normally, each device in a Linux system has its own queue and each queue can be tuned independently. However, the simulator assumes that there is a single device in the system (thus a single queue) that con-

tains all of the database objects. The current implementation is using a simulation of the stock readahead algorithm found in Linux 2.6 kernels [10]. Figure 2.7 depicts the decision flow diagram for a given 8KB I/O request from a request sequence.

After the cache simulations, timing information is associated with each I/O request. To do this, we use the query throughput $\mathcal{T}$ that is supplied as a parameter to the workload estimation process. Query throughput is first translated to I/O throughput by multiplying query throughput by the average number of I/O requests per query:

$$\mathcal{T}_{io} = \mathcal{T} \cdot \frac{\text{Total Number of I/O Requests in the Trace}}{\text{Total Number of Queries}}$$

Here, "Total Number of I/O Requests in the Trace" is the total length of the combined I/O request trace obtained after cache simulations, and "Total Number of Queries" gives the number of queries handled in the round-robin merging loop. The jth request in the representative I/O trace is assigned an arrival time of $j/\mathcal{T}_{io}$. This reflects the requirement that the necessary query throughput at the target operating point be satisfied by a storage system capable of handling I/O requests at this rate.

2.2.3 Fitting the Rome Model

To produce a Rome model of the I/O workload, a set of Rome stream parameter values must be chosen to characterize the I/O requests directed to each of the physical database objects. To select parameter values for a given object, first object's requests from the aggregate representative trace are projected, and then Rome parameter values (see Figure 2.2) are chosen to fit the per-object trace.

We take advantage of an existing I/O trace analysis tool called Rubicon [54] to implement this procedure. Rubicon implements both the per-object projection of the representative trace as well as the parameter fitting. Rubicon includes a number of statistical analyzers for estimating Rome model parameters from a request trace. Figure 2.8 summarizes how each of the per-object Rome parameters is estimated. In the figure, "I/O Count" of an object is the total number of I/O requests directed at the object.

2.3 Experimental Evaluation

In this section, an empirical evaluation of the proposed storage workload estimation technique will be presented. The evaluation has two goals: to determine how accurate the storage workload estimates are and to analyze how costly it is to generate those estimates.

Here, the important question is how to measure accuracy. One way to characterize accuracy is to compare, for a given database workload, the estimated storage

Symbol	Description	Estimation
t_{on}	average burst duration	Trace requests are partitioned into bursts, with a request interarrival gap greater than 2 seconds indicating a burst boundary. t_{on} is estimated as the *average* duration of the resulting bursts.
t_{off}	avg. inter-burst gap duration	Estimated as the *average* duration of the inter-burst gaps.
λ^r	read request rate during bursts	Estimated as the I/O count divided by the sum of the burst lengths.
λ^w	write request rate during bursts	Estimated as the I/O count divided by the sum of the burst lengths.
B^r	average read request size	Estimated as the *average* size of the read requests in the trace.
B^w	average write request size	Estimated as the *average* size of the write requests in the trace.
Q	avg. sequential run length	Trace requests are partitioned into runs. Consecutive requests are part of the same run if the starting position plus the length of the first request matches the starting position of the second request. Otherwise, there is a run boundary between the requests. Q is estimated as the *average* length of the runs in the trace.
$\Phi_i[j]$	pairwise overlap fraction	This describes how stream j's bursts overlap with those of stream i, $1 \leq j \leq \|\mathcal{D}\|$. First, the total amount of time during which both streams are simultaneously in I/O bursts is measured. This is divided by the sum of stream i's burst durations to generate an estimate for $\Phi_i[j]$.

Figure 2.8: Estimation of Rome I/O Model Parameters Using Rubicon Trace Analyzers

workload with the actual storage workload generated by the DBMS. This approach gives a characterization of accuracy that is independent of the intended usage of the storage workload estimate. However, it requires that we have some means of comparing storage workloads, which are complex artifacts.

An alternative means of evaluation is to characterize the suitability of the estimated workload for a particular purpose. In this case, the primary interest is in generating storage workload characterizations that will be useful as input to design and configuration advisors for storage systems. Such advisors use storage system cost models to determine how well a particular storage system configuration will perform under a given workload. Thus, one way to characterize the accuracy of a workload estimate is to use both estimated and actual workloads as input to a storage system performance model, and test whether they result in similar performance predictions. If they do, this indicates that the estimated workloads are accurate

enough to replace actual workloads as inputs to a storage system design advisor.

Both types of evaluation have been considered. In Section 2.3.2, we present a direct comparison of estimated and measured workloads. In Section 2.3.3, we examine the utility of estimated workload traces for the purpose of predicting the performance of various storage system configurations. Section 2.3.4 presents measurements of the cost of estimation.

2.3.1 Experimental Configuration

The Linux kernel-2.6.21.7's low-level block IO layer was modified so that it would produce I/O request logs when the DBMS is running. The request logs include one record for each I/O operation initiated by PostgreSQL. Since I/O operations are logged at I/O scheduler queue level, the request logs capture PostgreSQL I/O requests that miss both the database buffer cache and the VFS cache. They also accurately reflect writeback delays and readahead I/O requests. These logs are the actual storage workload generated by the database system, against which the estimated workloads can be compared.

PostgreSQL 8.0.6 is used as the DBMS. It was running on a Dell Poweredge 2600 server with two 2.2 GHz Intel Xeon processors and 4GB of main memory, running SUSE 10.0 Linux kernel-2.6.21.7. The server has a 70GB 15K RPM SCSI disk that is used to hold all system software, including PostgreSQL itself, as well as the I/O logs. In addition, the server has four 18.4GB 15K RPM SCSI hard drives behind a configurable Dell Perc 4Di RAID controller. These drives were configured into a single RAID0 LUN (logical device), on which a Reiserfs file system was built to hold the database objects.

PostgreSQL uses an 8KB page size, and the database system is configured with a 2GB shared buffer. The PostgreSQL `work_mem` parameter, which controls the amount of memory used by operators that hash or sort, was set to 2MB. The kernel's I/O scheduler queue properties for the logical device storing the database objects were set as follows: max_sector_kb to 128KB, readahead_kb to 1024KB, scheduler type to noop. Note that, since the storage subsystem has a RAID controller which employs I/O re-ordering algorithms, noop scheduler gives the best performance results among all four available scheduler types. The kernel's virtual memory parameter settings were kept at their default values: dirty_ratio to 40, dirty_writeback_centisecs to 500, and dirty_expire_centisecs to 3000.

We experimented with a scale-factor 5 TPC-H database. We used the open source implementation of the TPC-H benchmark [40]. Normally, the TPC-H benchmark [15] defines 22 query types, but Query 9 and Query 15 were excluded from our experiments. Query 9 was not included because of its excessive run time compared to the run times of other queries. While the average run time of other queries is around 10 minutes, the run time of Query 9 is around 260 minutes in our system. Query 15 refers to a view, which is not supported by the current implementation of the I/O Estimator. Thus, the SQL workload running against the TPC-H

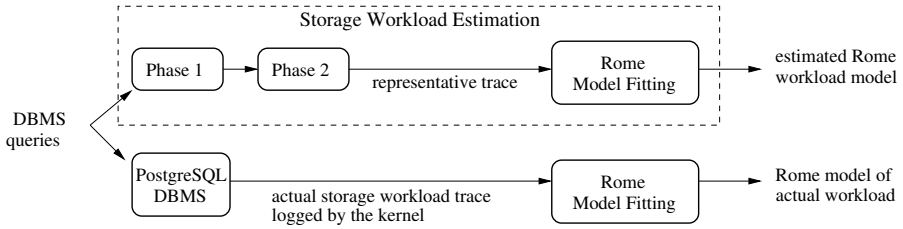


Figure 2.9: Experiment Design for Comparing Estimated and Measured Storage Workloads

database consisted of randomly-generated instances of 20 TPC-H query types with each query type having equal probability of occurrence. The execution plans for these queries make use of a total of twenty physical database objects, including 8 tables (lineitem, orders, supplier, customer, part, partsupp, region, and nation), 4 primary key indexes (orders_pkey, supplier_pkey, customer_pkey, and part_pkey), 7 indexes (i_l_orderkey, i_l_suppkey_partkey, i_l_suppkey, i_l_partkey, i_o_custkey, i_o_orderdate, and i_ps_partkey) on lineitem, orders and partsupp tables, and a tablespace for temporary files (TempSpace). This workload will be denoted by `WTPCH`.

2.3.2 Accuracy of Estimated Workloads

The first goal was to directly compare estimated storage workloads with actual storage workloads logged by the kernel. Figure 2.9 illustrates the design of the experiment. Using the default `WTPCH` query workload at a specified multiprogramming level `CL`, queries were generated and submitted to PostgreSQL for execution. A trace of the actual storage (I/O) workload generated by PostgreSQL as it executed the queries was captured, and the query throughput, $\mathcal{T}$ was also measured. We then used this database workload, including the measured query throughput, as input to the storage workload estimator.[1] This produces an estimated storage workload model, M_{est}. Finally, a Rome workload model was fitted to the actual storage workload trace using the same Rubicon-based model fitting procedure used in the third phase of the I/O estimation process. This results in a Rome model of the measured storage workload, which is denoted by M_{meas}. Appendix B gives a direct comparison of all of the I/O workload parameters obtained from the actual workload trace and the estimated representative trace for the `WTPCH` workload at two concurrency levels, `CL` $= 1$ and `CL` $= 5$. Here, we present a summary of those comparisons.

Figure 2.10 summarizes the results as weighted average estimation errors over all of the objects' request streams. Error is computed for each database object

[1]It was ensured that the estimator ran the same set of queries as PostgreSQL, and that they were initiated in the same order as they were in PostgreSQL. This ensures that any storage modeling error can be attributed to the proposed methodology and not to differences in the database workloads seen by PostgreSQL and the I/O estimator.

Workload	I/O Size (B^r, B^w)	Burst Request Rate (λ^r, λ^w)	Burst Time (t_{on})	Inter-Burst Time (t_{off})	Run Count (Q)
$CL = 1$	15%	45%	65%	39%	18%
$CL = 5$	33%	64%	106%	57%	29%

Figure 2.10: Weighted Relative Estimation Errors. Error is computed for each database object as the absolute difference of the estimated and measured values, divided by the measured value. The reported value is a weighted average over all database objects, with weights determined by the objects' measured total I/O counts.

	Percentage Burst Time $\frac{t_{on}}{t_{on}+t_{off}}$	Average Request Rate $\lambda \cdot \frac{t_{on}}{t_{on}+t_{off}}$
$CL = 1$	16%	14%
$CL = 5$	9%	44%

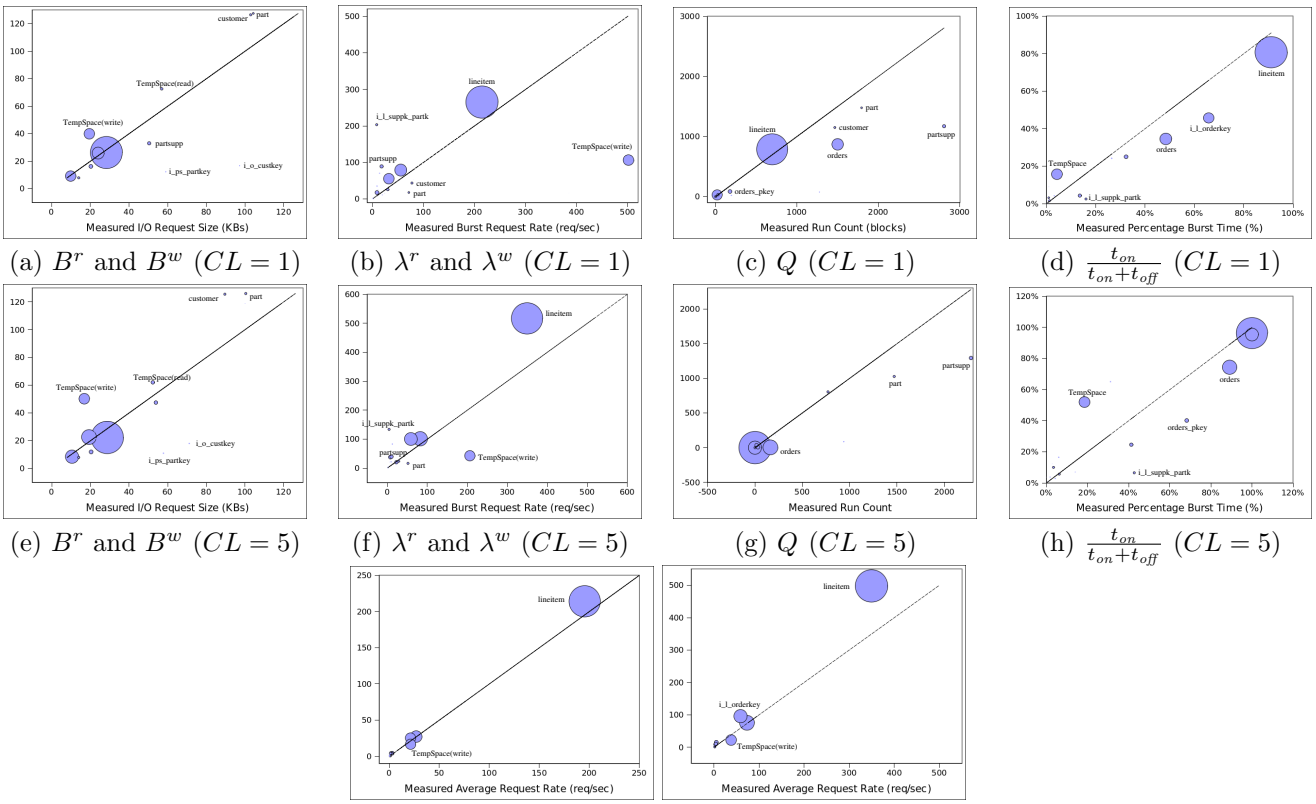
Figure 2.11: Weighted Relative Estimation Errors for Percentage Burst Time and Average Request Rate.

as the *absolute* difference of the estimated and measured values, divided by the measured value. The weights are determined by the objects' *measured* I/O counts. In addition to the parameters that are explicitly specified in the Rome model, we find it useful to compare some parameters that can be derived from the existing Rome parameters. These parameters are "percentage burst time" and "average request rate", which are computed as follows

$$\begin{aligned} \text{Percentage Burst Time} &= \frac{t_{on}}{t_{on}+t_{off}} \\ \text{Average Request Rate} &= \lambda \cdot \frac{t_{on}}{t_{on}+t_{off}} \end{aligned}$$

"Average request rate" is an important parameter to consider for two reasons. First, it indicates how accurately we estimate the I/O count of an object because it approximates the request rate as if the stream is always active. In other words, it ignores the burstiness of the request stream. Second, as will be introduced in Section 2.3.3, the storage system performance model utilizes average request rate rather than burst rate. Figure 2.11 summarizes the comparison of estimated and measured values for these two parameters as weighted average estimation errors over all of the objects' request streams.

Figure 2.12 compares some of these statistics in measured (M_{meas}) and estimated (M_{est}) workload models and the derived parameters described above. Each bubble in these graphs represents one database object, with bubble sizes scaled to the object's measured I/O count. Thus, more important objects have larger bubbles. Note that TempSpace is the only database object which receives write I/O requests. Therefore, we will not present two separate graphs for the parameters that are distinguished by I/O type; these parameters are B, λ and λ_{avg}. Instead, in the graphs



(a) B^r and B^w $(CL = 1)$ (b) λ^r and λ^w $(CL = 1)$ (c) Q $(CL = 1)$ (d) $\frac{t_{on}}{t_{on}+t_{off}}$ $(CL = 1)$

(e) B^r and B^w $(CL = 5)$ (f) λ^r and λ^w $(CL = 5)$ (g) Q $(CL = 5)$ (h) $\frac{t_{on}}{t_{on}+t_{off}}$ $(CL = 5)$

(g) λ^r_{avg} and λ^w_{avg} $(CL = 1)$ (h) λ^r_{avg} and λ^w_{avg} $(CL = 5)$

Figure 2.12: Estimated vs. Measured I/O Workload Model Parameters. Values from M_{est} are on the vertical axis, values from M_{meas} are on the horizontal axis. The center of each bubble represents estimated and measured values for one database object. Bubble area is scaled to the corresponding object's total I/O count. The line in each graph is estimated = measured, indicating perfect estimation.

of these parameters, we will represent TempSpace using two objects, one for read requests (TempSpace(read)) and the other for write requests (TempSpace(write)).

Before analyzing the statistics parameter-by-parameter, it is worth noting possible sources of errors in our estimations. The simplifying assumptions made in data-free plan simulation are one source of estimation errors. One of the other important reasons behind these errors is inaccurate query optimizer statistics and estimations. For example, TempSpace object is an outlier for the parameters which directly depend on I/O count (see Figure 2.12(b),(d),(f) and (h)). It is no surprise that the PostgreSQL version we used has made inaccurate cardinality estimates pertaining to this object. For instance, Figure 2.13 depicts the execution plan generated by PostgreSQL for TPC-H Query 18. While the optimizer estimates that `materialize` operator will create a temporary object for 30 million tuples, in the actual run the operator materializes only 36 tuples. Note that inaccurate cardinality estimations further affect the access pattern to the buffer cache in our simulations. For example, overestimating the I/O count of an object may result in

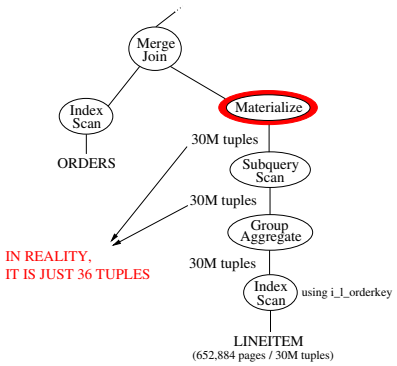


Figure 2.13: Execution Plan for TPC-H Query 18

evicting pages that would be utilized by an upcoming query. Comparing `CL` = 1 and `CL` = 5 cases in Figure 2.12, we see that capturing the caching impact on an I/O workload has a crucial role. Normally, as the concurrency level increases, one would expect the estimation errors (due to inaccurate cardinality estimates) to get worse. However, we do not observe a dramatic increase in error rates at $CL = 5$ (see Figures 2.10 and 2.11). Although error rates tend to increase as the multiprogramming level increases, one of the reasons is the difference between query mixes for $CL = 1$ and $CL = 5$. At $CL = 1$, the database workload is composed of 40 queries, a random mix of 20 TPC-H queries and each query template occurs twice in the mix. At $CL = 5$, the database workload is composed of 60 queries, a random mix of 20 TPC-H queries and each query template occurs three times in the mix. As a result, query templates with erroneous cardinality estimates, such as Query 18, occur more at $CL = 5$. The overall effect is that the estimator overestimates the number of I/O requests more at $CL = 5$ than it does at $CL = 1$. In the following, we discuss some of the other reasons behind the estimation errors.

Request Size: Both at $CL = 1$ and $CL = 5$, the highest error rates are observed for TempSpace object and for certain indexes (see Figure 2.12(a) and (e)). The I/O size is overestimated for TempSpace because of our assumptions. For example, a sort operator uses a single temporary file when performing merge-sort algorithm. It stores initial and intermediate sorted runs in this file. When simulating sort operator we assume that each run is accessed sequentially; however, in reality, this may not be the case. In our simulations, sequentiality triggers readahead which in turn results in I/O requests larger than 8KB. As for the indexes, I/O sizes are mostly underestimated for certain indexes. For example, while the measured I/O size is 97KB for `i_o_custkey`, it is estimated 17KB (see Figure 2.12(a)). This behaviour is observed for the indexes that are used in a subplan in a query execution plan. Such subplans are executed repeatedly for a given value, generally to check whether the value exists in the underlying relation. When we simulate a subplan, we always re-initialize its state at each call (i.e., getNext()) and thus pick a random offset

within the index (see Figure 2.5). Therefore, our simulation does not take into account any locality which may be observed in an actual run. Note that access locality may trigger readahead, and thus increase the I/O size. In addition, we do not simulate request merging that happens at the operating system's I/O queue so access locality again may result in request merging that we ignore in our simulations. Nevertheless, the weighted average error for the I/O size parameter is small: 15% for $CL = 1$ and 33% for $CL = 5$ (see Figure 2.10).

Burst Time and Percentage Burst Time: Burst time (t_{on}) and inter-burst gap time (t_{off}) are the two Rome parameters for which we observe the highest error rates (see Figure 2.10). As was presented in Figure 2.8, t_{on} and t_{off} are the average duration of all burst and inactive periods, respectively. Although we make large errors in estimating the average duration of the burst period of a stream, we estimate percentage burst time much more accurately (see Figure 2.11). As a matter of fact, percentage burst time is more important for analytical storage system models. As will be explained in the following section, the storage system performance model utilizes percentage burst time rather than burst duration. t_{on} and t_{off} are crucial for trace-based storage system simulators [59], in which a Rome stream is simulated as if it is an infinite I/O stream and active for t_{on} with a period of t_{off}. Burst periods play the most important role when analyzing two streams' relative active periods; Rubicon uses burst periods to compute the pairwise overlap fraction matrix Φ. As we will discuss shortly, the I/O estimator does a good job in estimating how two streams' burst time overlaps. In conclusion, although we make large errors in estimating t_{on} and t_{off}, the I/O estimator accurately estimates the relative duration of burst and inter-burst times of a stream and the relative burst times (i.e., correlation) between two streams.

Burst and Average Request Rates: Rubicon computes the burst rate as $\frac{\text{I/O Count}}{\sum t_{on}}$ (see Figure 2.8). Thus, any error made in the estimation of the objects' I/O counts would be reflected into the request rate estimations. Moreover, as described in Section 2.2.2, I/O request time is computed using query throughput $\mathcal{T}$ and total estimated I/O count, thus $\sum t_{on}$ depends on I/O count, too. The I/O estimator tends to overestimate the burst request rate for most of the objects (see Figure 2.12(b) and (f)) because for both `CL` $= 1$ and `CL` $= 5$ I/O counts of those outlier objects are overestimated. For example, while the measured I/O count of `lineitem` table is 4.6 million and 6.2 million requests for `CL` $= 1$ and `CL` $= 5$ respectively, those are estimated at 5.1 million and 7.8 million requests. However, the estimation errors for the *average* request rate is lower (see Figure 2.11) because it approximates I/O count and cancels out the impact of $\sum t_{on}$.

Run Count: The estimator is able to capture the effect of concurrency on sequentiality (see Figure 2.12(c) and (g)). For example, while lineitem and

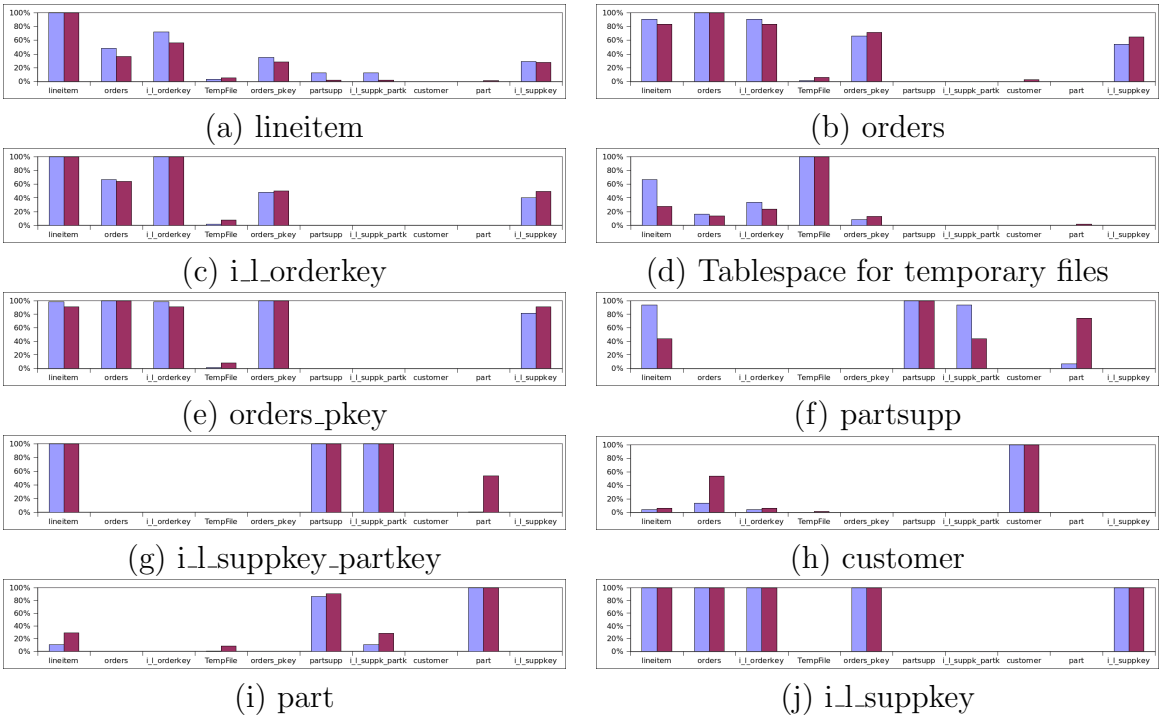
Object	$CL = 1$	$CL = 5$
lineitem	5.9%	16.6%
orders	3.7%	13.9%
i_l_orderkey	2.1%	16.5%
TempSpace	5.8%	14.3%
orders_pkey	3.1%	12%
partsupp	16.8%	17.7%
i_l_suppkey_partkey	5.2%	14.8%
part	4.8%	9%
customer	4.6%	23.5%
i_l_suppkey	0%	11.3%
Weighted Estimation Errors over all Objects	5.4%	15.9%

Figure 2.14: Overlap Fraction Average Absolute Estimation Errors. Error is computed for each database object as the absolute difference of the estimated and measured values, divided by the number of objects. The last row is the weighted average of these errors over all objects, with weights determined by the objects' measured total I/O counts.

i_l_orderkey objects are accessed sequentially at `CL` = 1, they are being randomly accessed at `CL` = 5. There are a few outliers such as part and partsupp tables, but these are already sequentially accessed at both `CL` = 1 and `CL` = 5. Fortunately, this kind of run length estimation error is probably not significant. Any Q greater than 100 represents very sequential I/O. The most important thing for the estimator is to distinguish between very small run lengths (e.g., $Q = 1$) and large run lengths. In Section 2.3.3, we show that these large run length estimation errors do not lead to large errors in predicting the performance of the underlying storage system.

The remaining I/O model feature that has not been discussed yet is the burst overlap matrix, Φ. Figures 2.15 and 2.16 compare measured and estimated burst overlaps, $\Phi_i[j]$, among the database objects' request streams for `CL` = 1 and `CL` = 5. Each graph shows the percentage overlap of one object's request bursts with the bursts of other objects. The figure includes only the ten objects with the highest measured I/O counts. Figure 2.14 summarizes these results as the average absolute estimation errors over the overlap fractions of each object with all of the remaining objects. Error is computed for each database object as the *absolute* difference of the estimated and measured values, divided by the number of objects: for object $object_i$, $\frac{\sum_{k=1}^{|\mathcal{D}|} |\Phi_i[k]_{meas} - \Phi_i[k]_{est}|}{|\mathcal{D}|}$. The last row in Figure 2.14 summarizes these per-object estimation errors as the weighted average estimation error over all of the objects, and the weights are again determined by the objects' measured I/O counts.

These data show that the estimator does a good job of estimating which objects' request bursts overlap and which do not in both `CL` = 1 and `CL` = 5. For example, at `CL` = 1, when the orders table is being accessed (Figure 2.15(b)), so are the lineitem table, the orders table's primary index (orders_pkey), and the i_l_orderkey



(a) lineitem (b) orders

(c) i_l_orderkey (d) Tablespace for temporary files

(e) orders_pkey (f) partsupp

(g) i_l_suppkey_partkey (h) customer

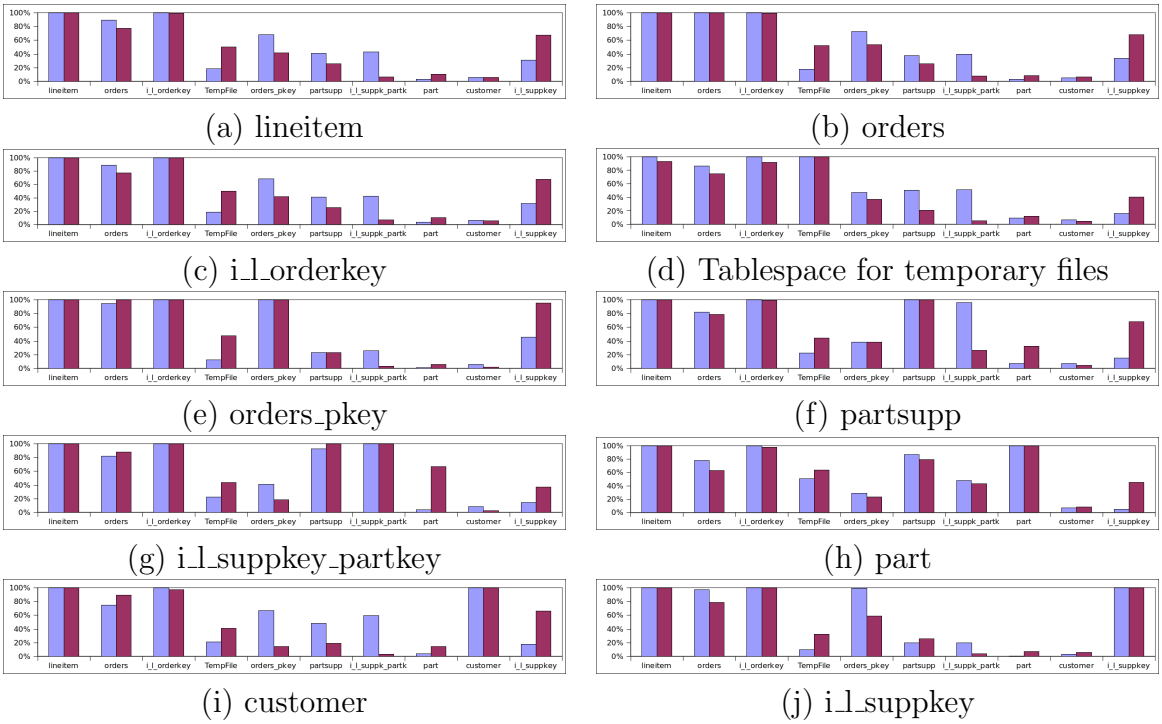
(i) part (j) i_l_suppkey

Figure 2.15: Measured and Estimated Burst Overlaps at CL = 1. The left (blue) bar of each pair shows measured values, the right (red) bar shows estimated values.

and i_l_suppkey indexes. Other objects, like the part, partsupp and customer tables, are not. This kind of information is useful to storage configuration tools, since placing co-accessed objects on different storage devices can reduce interference and improve performance [2]. As the concurrency increases, some objects start to be co-accessed. When the partsupp table is being accessed at CL = 1 (Figure 2.15(f)), some objects like orders table, i_l_orderkey and orders_pkey indexes and TempSpace object are never accessed. However, at CL = 5 (Figure 2.16(f)), those objects are accessed.

2.3.3 Storage Performance Prediction Using Estimated Workloads

As was noted in the beginning of the chapter, one of the motivations for generating storage workload estimates is to be able to use them to estimate the performance of candidate storage configurations. Storage configuration advisors, such as the one that will be introduced in the following chapter, use storage system models to estimate the performance of candidate configurations. In this section, we present the results of an experiment in which both measured and estimated storage workload models were used as input to a storage system performance model. We then compared the storage system performance predicted by the model under the two workloads. Ideally, the predictions would be identical. This would indicate that the



(a) lineitem (b) orders

(c) i_l_orderkey (d) Tablespace for temporary files

(e) orders_pkey (f) partsupp

(g) i_l_suppkey_partkey (h) part

(i) customer (j) i_l_suppkey

Figure 2.16: Measured and Estimated Burst Overlaps at CL = 5. The left (blue) bar of each pair shows measured values, the right (red) bar shows estimated values.

estimated workload models were as good as the measured models for this particular task.

For this experiment, we used the storage system performance model that will be explained in detail in the following chapter. This performance model takes Rome storage workload descriptions as input. Thus, the output of the storage workload estimator can be used without modification as input to this model. The model assumes that a storage system is structured into a set of independent RAID groups (or arrays), and it describes how the "stores" that are referred to in the workload are mapped to RAID groups. In our case, each store corresponds to a physical database object, so the layout effectively describes how database objects are laid out on the available RAID groups. The storage system performance model predicts the average utilization of each RAID group for a given set of objects, their I/O workloads (i.e., Rome streams) and mappings to the RAID groups. Although workloads are bursty, the model predicts average utilization rather than peak utilization by using average request rate which is derived from burst rate and percentage burst time as described in the preceding section. The evaluation of this storage system performance model is presented in the next chapter, and the empirical results show that the model estimates device utilizations accurately.

To run each experiment, first a target storage system configuration must be chosen. We have designed different RAID group configurations and object layouts. RAID groups and layouts are *not* materialized in our experimental environment

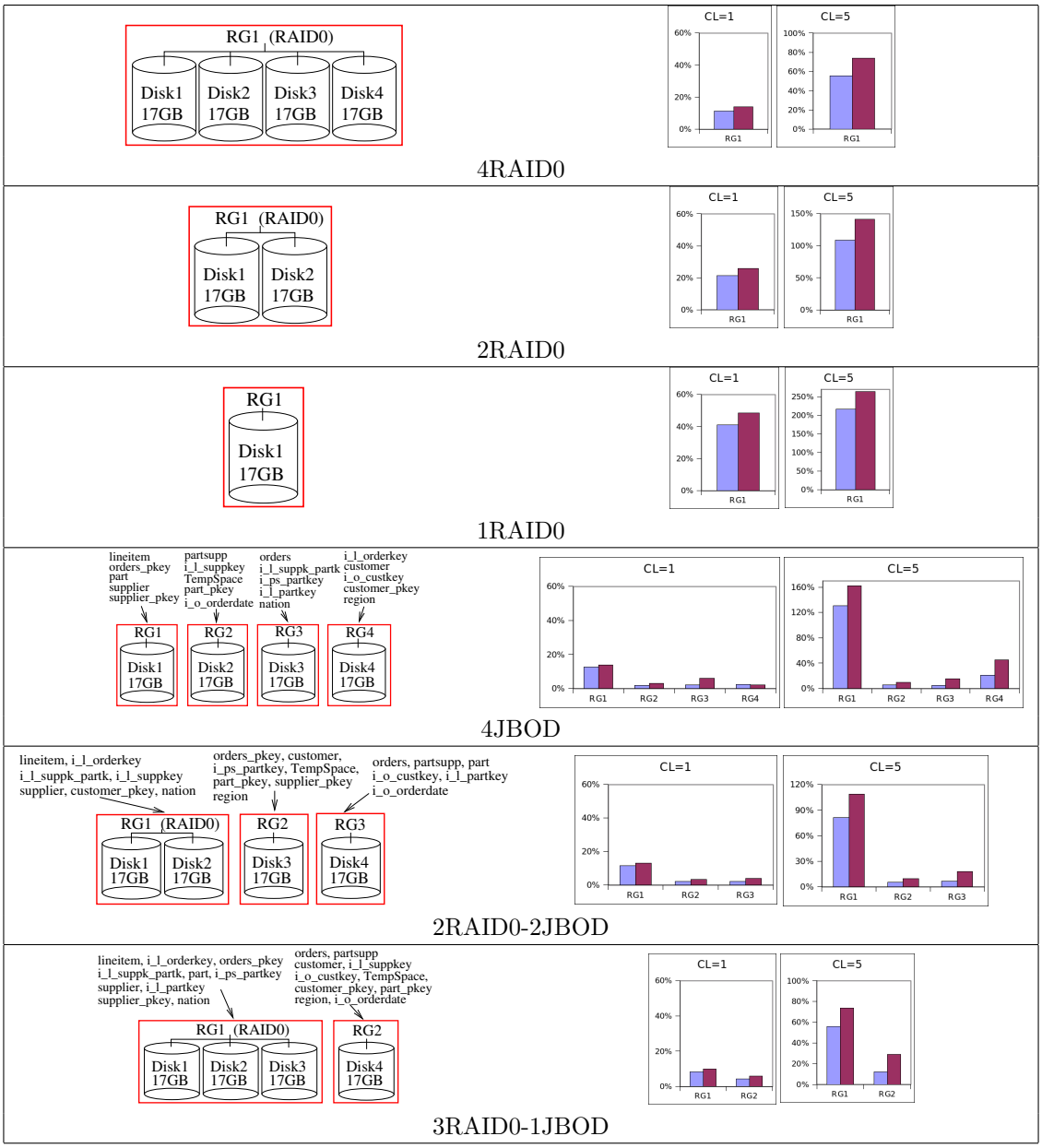


Figure 2.17: Storage System Performance Predictions Under Estimated and Measured Storage Workload Models. Each storage system configuration is illustrated next to a graph showing the average percentage utilization of each RAID group. In the graphs, the left bar in every pair shows utilization under M_{meas}, and the right bar shows utilization under M_{est}.

but their descriptions are given inputs to the storage system performance model as "what-if" storage system configurations. The measured and estimated storage workload models obtained for `WTPCH` at `CL` $= 1$ and `CL` $= 5$ are fed to the storage system model for each of the target configurations. Figure 2.17 illustrates the storage system configurations that we tested, as well as the predicted storage system

Wkld	Phase1	Phase 2 (Merging) DB Caching	VFS Sim.	Timestamp	other	Phase3	Total
CL = 1	280	77	500	52	40	35	984
CL = 5	280	119	626	83	59	58	1225

Figure 2.18: Total and Per-Phase Times (in seconds) for Storage Workload Estimation of WTPCH SQL workload.

utilizations under M_{meas} and M_{est} for CL = 1 and CL = 5. Three of the configurations (4RAID0, 2RAID0, 1RAID0) defined a single RAID0 group (labeled RG1 in Figure 2.17) to store all database objects. The other three configurations (4JBOD, 2RAID0-3JBOD and 3RAID0-1JBOD) defined multiple RAID groups. For those configurations, Figure 2.17 specifies which physical database objects were mapped to the each RAID group. Basically, objects are distributed across RAID groups in a round-robin fashion starting from the object with the highest measured I/O count to the one with the least.

Figure 2.17 shows that estimation errors in RAID group utilizations are mostly small. It also shows that RAID group utilizations predicted using M_{est} tend to be higher than the ones using M_{meas}. It is mostly because the request rates of important objects are overestimated as depicted in Figure 2.12(g) and (h). Although M_{est} gives higher estimates, it accurately captures the relative differences between RAID group utilizations that are observed when M_{meas} is used for all of the configurations. Note that the 4JBOD, 2RAID0-2JBOD and 3RAID0-1JBOD configurations used data layouts that are quite imbalanced, and that this imbalance is easily observable in the performance predictions produced with M_{est}. This is a limited experiment with a small number of storage system configurations. However, we consider that predictions of this level of accuracy are quite reasonable, given that we start with SQL workloads. For storage administrators, the alternatives (of populating the database, running a workload, and measuring the resulting storage system load) are guesstimates and rules of thumb.

2.3.4 Cost of Storage Workload Estimation

Figure 2.18 shows the wall-clock time required for storage workload estimation under our two workloads. Phase 1 takes the same amount of time for both experiments because both do exactly the same thing: generate I/O request sequences for 20 TPC-H query types. Phase 2 includes the request sequence merging, database buffer cache simulation, and VFS cache, readahead and writeback simulations. Phase 2 spends most of the time for the VFS simulation. As a matter of fact, VFS simulation is the most costly part of the whole simulation process, and accounts for 51% of the total simulation time for both $CL = 1$ and $CL = 5$. In the current version of the estimator, some inefficient data structures were used when implementing the VFS simulation. Therefore, it is is open for optimization. Re-

ducing the run time of the VFS simulation will reduce the run time of Phase 2. Finally, Phase 3 is the statistical analysis performed by Rubicon. These times depend primarily on the total length of the representative storage workload trace that is generated. The representative trace is composed of 6.5 million and 9.7 million I/O requests for $CL = 1$ and $CL = 5$, respectively. Measurement times for `WTPCH` at $CL = 1$ and $CL = 5$ are 24000 and 18000 seconds, respectively. A measurement duration can be compared with the summation of Phase 1 and Phase 2, which is 949 seconds for $CL = 1$ and 1167 seconds for $CL = 5$. As a result, data-free simulation is 25 times faster for $CL = 1$ and 15 times faster for $CL = 5$ than the corresponding real executions.

2.4 Conclusion

We have presented a technique for estimating the storage system workloads that are generated by database management systems. The technique generates storage workload models in a form that is easily used by storage administration tools, such as configuration advisors. The feasibility of this approach has been demonstrated by implementing it in PostgreSQL DBMS. The experimental results suggest that the workload estimations produced by the proposed technique are sufficiently accurate to be useful for predicting the performance of alternative storage configurations. We expect the estimates to be of similar use for other related tasks, such as capacity planning. This is the first attempt that we are aware of to design tools intended to improve the flow of information from the database tier to the storage tier.

Chapter 3

Workload-aware Storage Layout for Database Systems

The preceding chapter has presented a technique for estimating the I/O workload that a database system will generate for a given SQL workload. I/O workload characterizations can also be obtained by monitoring an active DBMS and obtaining its I/O trace. In this chapter we will describe how I/O workload information, no matter whether it is obtained through estimation or monitoring, can be leveraged to reach a storage layout design which is tuned to that workload. The performance of a database system depends on the layout of its objects, such as tables or indexes, on the underlying storage devices.

In the following sections, we first discuss what exactly storage layout for database systems means. Later, current practices, which are used by system administrators while laying out data, will be presented. Then, a formal definition of the layout problem will be introduced. The presented approach to the problem requires exploiting information about the I/O workload and the underlying storage system, so a way to model I/O workload and the storage system is required. Therefore, after giving the formal definition of the problem, we will explain in detail how such models can be built. Afterwards, we present and evaluate the proposed layout algorithm. Lastly, potential extensions of this work will be discussed.

3.1 Database Storage Layout Problem

Database management systems rely on an underlying storage system for persistent storage of database objects such as tables, indexes and logs. The storage system also hosts temporary objects and intermediate results when primary storage (i.e., memory) is not sufficient. A storage system typically provides a set of RAID groups (groups of storage devices in some type of RAID configuration) or individual storage devices, such as rotational disk drives or solid-state drives (SSDs). We will refer to each distinctly addressable and independent data container as a *storage target*. At

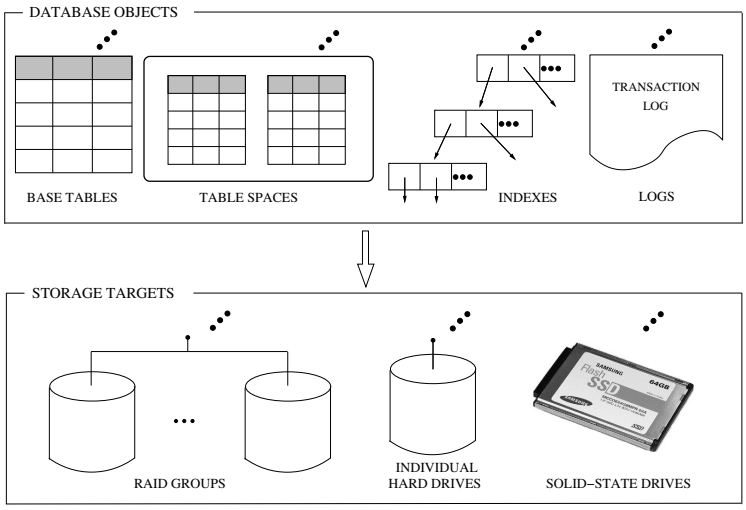


Figure 3.1: At the coarsest level, database storage layout problem is to find a mapping of database objects onto the available storage targets.

the coarsest level, the database storage layout problem can be seen as the problem of finding a mapping of database objects to the available storage targets (see Figure 3.1). In Section 3.1.2, when a formal definition of the problem is presented, more details on the nature of this mapping will be given.

Our goal is to improve the performance of a database system by providing a good layout. A good layout should result in a *balanced load* across the storage targets. Otherwise, the most heavily utilized target may become a performance bottleneck while storage bandwidth available from other targets is wasted. A good layout should *minimize interference* between requests for different objects that are laid out on the same target. For example, if a sequentially accessed object is laid out on the same target as some other objects (which are accessed either sequentially or randomly) and if all those objects are accessed simultaneously, then interference among I/O requests may prevent the underlying storage devices from exploiting the sequentiality, thus increasing data access time. Lastly, a good layout should *take into account the different characteristics of the available targets*. Storage targets may vary in capacity and performance. For example, RAID groups may vary in configuration, e.g., in the RAID level used, or in the number of devices in the group. Performance characteristics may also vary from device to device. Disk drives are typically added to storage systems over time, and the more-recently added disks are likely to be faster. Device 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 the one implemented using disk drives. The promise of good performance for database workloads with SSDs [31] 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 summary, the goal is to figure out a database storage layout which leads the storage system to have balanced load, minimized inter-object interference, and awareness of system heterogeneity. Unfortunately, these objectives may conflict with one another. For example, while balancing the load requires that each object be laid out on as many targets as possible, reducing interference necessitates isolating some objects from the others.

3.1.1 Current Practice

In current practice, database layout is primarily guided by heuristics and rules of thumb. Database vendors give generic advice to database administrators on how to design the storage backend for databases. Such advice includes tips on how many disk spindles to have per CPU core, how to configure storage targets out of disk spindles, and how to lay out data on those targets. For example, a vendor might recommend the use of at most 15-20 dedicated spindles per CPU core or the use of RAID10 targets for logs and RAID5 targets for data, or the allocation of 15%-25% of the disk spindles for logs and the rest for data. In this work, the primary focus is on how to lay out objects onto storage targets. Therefore, we will review only guidelines regarding layout.

Vendor advice is usually generic because database vendors assume minimal knowledge about the database I/O workload. As a result, they see simplicity as an important layout property. All major vendors suggest distributing every database object across all available targets [41, 47]; an approach that is known as *stripe-everything-everywhere* (SEE). Not only do vendors suggest SEE, but they also integrate it into their layout automation tools. For example, both IBM DB2 Automatic Storage (AS) and Oracle Automatic Storage Manager (ASM) implement the SEE approach[1]. SEE distributes data, thus workload, evenly across all available storage targets. As a result, it ensures that the load across targets will always be balanced. However, since SEE results in interference with sequentially-accessed objects, it may not result in an optimal performance for a given I/O workload. Moreover, to be able to get the most benefit from the SEE approach, it is required that the targets be uniform in capacity and performance [47]. Since all targets receive the same load under the SEE layout, faster targets will be under-utilized and the performance of the whole system will be bounded by the slowest target. As a result, it is not clear how to apply SEE effectively in heterogeneous systems.

[1] In Oracle ASM, the layout strategy is actually called *Stripe and Mirror Everything* (SAME). If economically feasible, Oracle advises mirroring in addition to striping. In SAME each device is mirrored, and thus the number of devices needed is doubled.

As discussed at the beginning of the section, it is not uncommon for a storage system to have storage targets varying in capacity and performance. Especially with the advent of SSDs, system heterogeneity has become a more commonly encountered situation. Therefore, DBAs face with the problem of deciding how to map database objects onto a pool of targets with different performance characteristics. The problem of figuring out which database object must be laid out on which target is notoriously viewed as an unachievable task [47]. However, database vendors offer heuristic layout hints in such situations, too. For example, they suggest what kinds of database objects to map to SSDs in order to get the most benefit out of them. The core idea of those heuristics is to isolate randomly accessed objects, such as unclustered indexes, onto SSDs [21]. Similar issue arises when the storage system has to serve different database servers, which may occur as a result of *storage consolidation*. Although database vendors recommend that objects belonging to different database servers be separated [47], it can be difficult to determine which targets to use for each database server's objects.

In conclusion, generic guidelines may give useful hints to DBAs on how to lay out database objects in case they have limited or no knowledge on the I/O workload, but in general it is not always clear how to apply these heuristic guidelines, especially in heterogeneous systems. Our layout technique goes beyond those generic guidelines by making use of a description of the database system's I/O activity to provide a workload specific layout. Thus, the underlying storage system can be tuned to the workload generated by the database(s) that use it. Moreover, this technique supports what-if scenarios and makes it feasible to examine non-trivial layouts. Commonly, testing a layout requires constructing the storage targets, materializing the database system and observing the active system's performance. Therefore, it is infeasible for a DBA to test a variety of layouts. However, the proposed approach enables a DBA to estimate the quality of a large set of possible layouts efficiently.

3.1.2 Formulation of the Layout Problem

Now, we will describe the database layout problem more formally. The software component which solves the layout optimization problem and recommends a good layout is called the *layout advisor*. We will explain the inputs expected by the layout advisor, and then present the formal definition of the layout optimization problem.

It is assumed that a storage system provides M disjoint storage targets, and c_j denotes the capacity of the jth target. The layout advisor is indifferent to the exact nature of the targets. No matter whether a target is a stand-alone storage device or a combination of devices, the layout advisor cares only that each storage target has an associated performance model and that the performance of each storage target is independent of the performance of the other targets. In an enterprise-class storage system [20, 24, 39], 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.

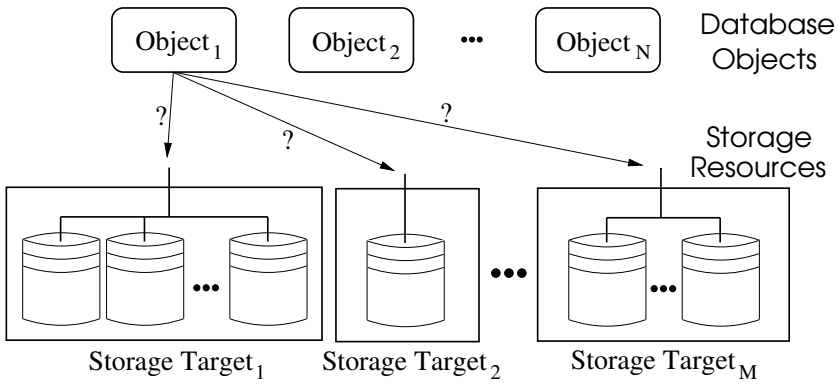


Figure 3.2: Database Object Layout Problem. Each database object is laid out onto a subset of available storage targets.

The layout advisor is given N disjoint database objects that are to be laid out on the available storage targets, and s_i denotes 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. Moreover, all of the objects may be part of the same database or they may originate from different databases. The layout advisor cares only that each database object has an associated set of I/O workload characteristics similar to the ones introduced in Chapter 2.

As stated earlier, database storage layout is a mapping of database objects to storage targets (see Figure 3.2). Thus, a *layout* L can be represented as an $N \times M$ matrix, where $0 \leq L_{ij} \leq 1$ is the fraction of object $object_i$ that is assigned to target $target_j$. Validity of a layout L is ensured through two constraints:

Definition 2 ***Capacity constraints:***

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

Definition 3 ***Integrity constraints:***

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

The capacity constraints ensure that no target is assigned more data than it can hold, and the integrity constraints ensure that each object is mapped in its entirety to the storage target(s).

The goal of the layout advisor is to identify a good layout from among all possible valid ones. Specifically, the advisor's objective is *to minimize the maximum utilization across all storage targets.* As a result, a layout L_i will be considered better than another layout $\mathrm{L}_j, j \neq i$, if the utilization of the most utilized target when

Symbol	Description
N	Number of database objects
s_i	Size of ith object
W_i	I/O workload for the ith object
M	Number of storage targets
c_i	Capacity of ith target
μ_j	Utilization of the jth storage target
L_{ij}	Layout: the fraction of object $object_i$ laid out on target $target_j$

Figure 3.3: Parameters of the Layout Problem

L_i is used is lower than the one when L_j is used. As was discussed, a good layout will allow the storage system to have certain characteristics: balanced load, minimized inter-object interference and awareness of system heterogeneity. Minimizing the maximum target utilization encourages the layout advisor to pick a good layout by balancing the potentially-conflicting optimization objectives described earlier. First of all, this objective encourages the advisor to identify balanced layouts, since the most heavily utilized target determines the quality of the layout. Secondly, it also encourages the advisor to avoid interference, since interference increases I/O request service times and hence storage target utilizations. Finally, when there are different types of storage targets, this objective 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.

Now, we will briefly explain what pieces of information the layout advisor uses to determine the storage target utilizations resulting from a candidate layout. More detailed explanation is deferred to the upcoming sections. Firstly, the advisor needs information about the I/O workload for each object. It is assumed that the layout advisor is given a *workload description* for each database object, and W_i is used to denote the workload description for the ith object. As described in the preceding chapter, an object's workload description characterizes 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. Section 3.2.1 will present the I/O workload model the layout advisor has adopted. Secondly, the advisor needs a storage system performance model, which predicts the utilizations of the targets for the given I/O workloads and a candidate layout. $\mu_j(W_1, \ldots, W_N, L)$ is used to denote the model-predicted utilization of the jth storage target under the I/O workloads $W_1, \ldots, W_N$ and the layout L. To simplify this 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. Section 3.2.2 will explain in detail the storage system performance model adopted by the layout advisor.

Figure 3.3 summarizes the parameters of the layout problem, and the following definition gives the formal problem statement.

Definition 4 ***Database Storage Layout Problem***
Given N database objects with sizes s_i, M storage targets with capacities c_j, 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 [41]. 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 implementation 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 across targets. This always distributes an object evenly across the targets. However, Definition 4 does not enforce an even distribution of the objects. For example, the layout of the object Table_T illustrated in Figure 3.4 is a valid layout according to this definition. For this reason, we provide the ability to optionally restrict the layout advisor to produce *regular* layouts:

Definition 5 ***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. In order to distinguish layouts which are *not* regular from the regular layouts, non-regular layouts will be named *general layouts* from now on. Although general layouts may not be materialized feasibly in some systems, there are commercial storage systems which favor this kind of layouts. For example, Network Appliances has recently introduced flexible volumes [17] for their storage servers (i.e., NetApp filers). A flexible volume, which can contain any number of data objects, can be mapped in any number of storage targets (i.e., RAID groups) available in a NetApp Filer and in any portion. The layout advisor is able to provide both general and regular layouts for a given problem instance. To be brief, it first finds out a general layout as the solution. If the target system can

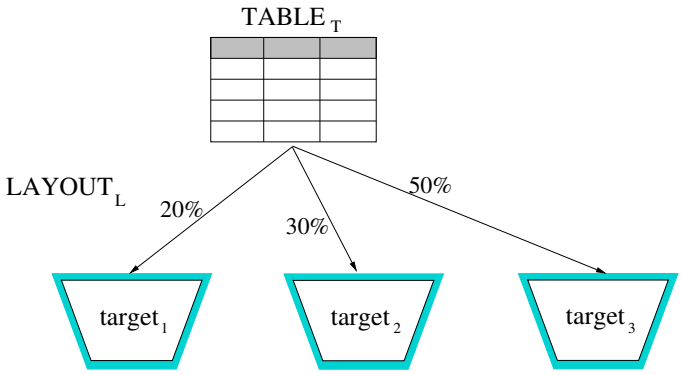


Figure 3.4: General (Non-Regular) Layout. Table T is laid out on three storage targets. In general layouts, objects can be distributed un-evenly across targets.

only implement regular layouts then the advisor converts the general layout into a regular one. The details of the layout algorithm adopted by the layout advisor will be discussed in greater detail in Section 3.3.

3.2 I/O Workload and Storage System Modeling

In this section, we will discuss the details of the database layout problem parameters, which have been briefly introduced in the preceding section. The layout advisor relies on workload descriptions (i.e., W_i's) and a storage system performance model which reports target utilizations (i.e., μ_j's), c.f. Definition 4. A workload description reveals certain characteristics of the I/O workload directed at a database object. An I/O workload model defines what characteristics are included in a workload description. A storage system performance model is used to evaluate the quality of a layout. Given a particular layout of objects and the I/O workload information for each object, this model has to estimate the utilization of the targets in the system. The maximum target utilization then determines the quality of the layout. What follows is the explanation of the workload and storage system performance models adopted by the current implementation of the layout advisor.

3.2.1 I/O Workload Model

I/O workload information acts like the glue between database and storage tiers. The workload information allows us to go beyond the generic guidelines often used by database administrators, and recommend a workload-aware layout. In Chapter 2, we have already introduced an I/O workload model to describe the characteristics of an I/O workload directed at a data object, i.e., the Rome model (language) [60, 61]. The layout advisor adopts the Rome model to describe the I/O characteristics of

Parameter		Description
Request Sizes	(B_i^r and B_i^w)	Average read/write request sizes
Request Rates	(λ_i^r and λ_i^w)	Average read/write request rates
Run Count	(Q_i)	Average number of requests in a sequential run
Overlap Fractions	($\Phi_i[j]$)	Temporal correlation of I/O requests in workload W_i with I/O requests from the jth workload, $1 \leq j \leq N$. Each workload description includes $N-1$ overlap statistics, one for each other workload.

Figure 3.5: I/O Workload Parameters. The layout advisor assumes that the I/O workload description W_i of object $object_i$ is composed of the above statistical parameters.

each database object because the Rome model's utility has already been demonstrated [5, 7, 8].

Remember that, in the Rome model, the I/O workload of a system is represented as a set of stores and a set of streams. Here, the layout advisor assumes that each workload description is a Rome stream, and each database object is a Rome store. The advisor makes use of a subset of the available Rome stream parameters, and Figure 3.5 summarizes those parameters included in each workload description W_i, $1 \leq i \leq N$. Thus, an I/O workload description is a set of $N + 4$ statistical parameters.

Note that a workload description captures the I/O workload characteristics for an object assuming that the object is accessed uniformly. However, database objects may observe skewed access; that is, some parts of an object may be accessed more frequently that the other parts. In order to capture the workload characteristics at this finer granularity, an object can be viewed as a set of stores (i.e., partitions) instead of a single store, each partition with its own workload description. Thus, partitions can be laid out independently.

3.2.2 Storage System Performance Model

The layout advisor needs a model that can predict the utilizations of storage targets, given descriptions of the workloads and a candidate layout. In theory, any function that maps workloads and layout to target utilizations could be used. In practice, the storage system model employed by the current implementation of the advisor has a particular internal structure, as illustrated in Figure 3.6. The model is composed of two sub-models.

As depicted in Figure 3.6(a), a *layout model* takes as an input an object's workload, W_i, and a candidate layout, L, and produces a description of the workload that will be generated by that object on each storage target. W_{ij} is used to denote the workload imposed by the ith object on the jth storage target under a given

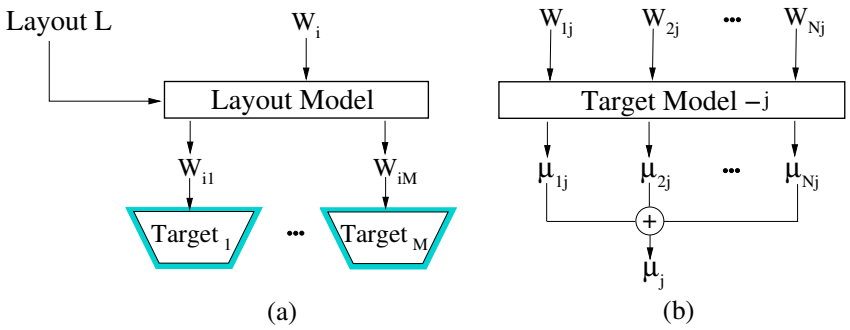


Figure 3.6: An Overview of the Storage System Model

layout. The layout model describes the distribution of the object's workload that is implied by the candidate layout. In brief, the layout model transforms each I/O workload W_i, $i = 1..N$, into sub-workloads W_{ij}, $i = 1..N$ and $j = 1..M$.

After all of the database object workloads have been distributed by the layout model, each storage target will have its own set of workload descriptions as illustrated in Figure 3.6(b). Given these workloads, a *target model* has to produce an estimate of the utilization of the storage target. The layout advisor, after obtaining the utilizations of all targets from the target models, represents the quality of the candidate layout by the maximum of these utilizations. In other words, the performance of the storage system under the given layout is represented by the utilization of the most utilized target. What follows is the explanation of the layout and target models adopted by the layout advisor.

3.2.2.1 Layout Model

A layout model describes how to transform the workload description of an object into per-target workload descriptions, given a particular layout. This 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. The per-target workload descriptions use the same Rome-style statistical parameters as the original object workload descriptions (see Figure 3.5). The layout model describes how to calculate the parameters of the per-target workload descriptions from the parameters of the database object workload descriptions and the layout. Development of such a model requires some knowledge of the mechanism that will implement the layout. In the current implementation, it is assumed that if an object is found on more than one target then it is striped across these targets. For example, to model the run count of W_{ij}, it is necessary to know the run count of W_i and the size of the "stripes" into which objects are broken when they are laid out onto the storage targets. Here, the stripe unit is a property of the layout mechanism.

Figure 3.7 is an illustration of the transformations of statistical parameters

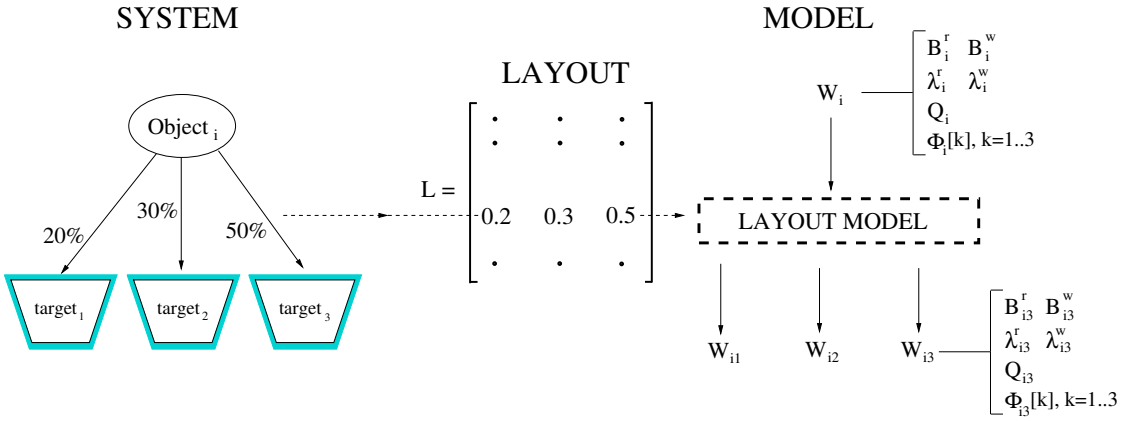


Figure 3.7: An Illustration of the Layout Model Transformations

applied by the layout model based on a given layout. For the experiments reported in Section 3.4, 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. In the following, we describe how the layout model estimates the parameters of the target workloads using the parameters of the given object workload.

Request Size:

$$B_{ij}^r = \begin{cases} B_i^r, & L_{ij} > 0 \\ 0, & \text{otherwise.} \end{cases}$$

$$B_{ij}^w = \begin{cases} B_i^w, & L_{ij} > 0 \\ 0, & \text{otherwise.} \end{cases}$$

A simplifying assumption here is that a request is not broken into smaller requests because of striping. Thus, the original I/O workload of an object and the sub-workloads directed at the targets containing this object all have the same request sizes.

Request Rate:

$$\lambda_{ij}^r = \lambda_i^r \cdot L_{ij}$$
$$\lambda_{ij}^w = \lambda_i^r \cdot L_{ij}$$

Since it is assumed that request size of an I/O workload does not change when the workload is striped across targets, request rate is transformed solely based on the layout. Additionally, it is assumed that the portion of an object laid out on a target will receive I/O requests proportional to how much of the object is mapped on that target. For example, given in Figure 3.7, 20% of object $object_i$ is residing on the target $target_1$. Based on the assumption, target $target_1$ will receive 20% of the requests directed at object $object_i$. Therefore, the request rate of the sub-I/O workload imposed by object $object_i$ on target $target_1$ will be 20% of the original workload's request rate: $\lambda_{i1}^r = 0.2 \cdot \lambda_i^r$.

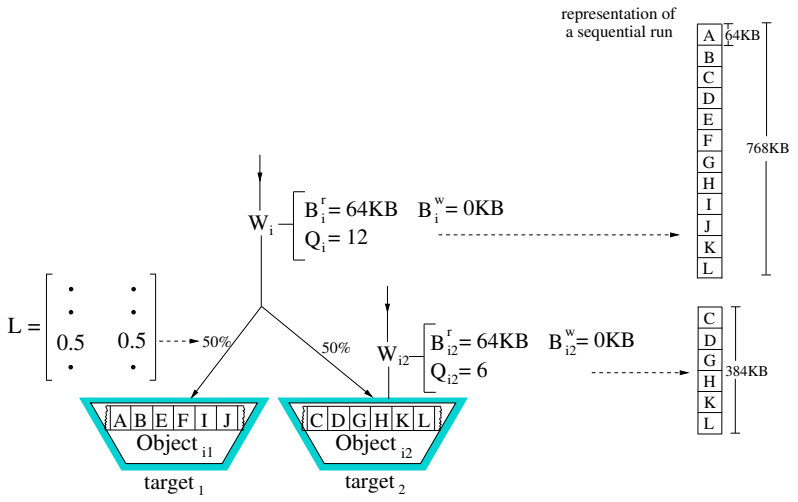


Figure 3.8: Layout Model: Run Count Transformation. A sequential run may be broken into smaller runs when object $object_i$ is striped across more than one target.

Run count: Run count parametrizes the sequentiality of an I/O stream (see Section 2.1.2 for more information). It represents the average number of I/O requests in a sequential run. A sequential run is a series of I/O requests sequentially accessing an object. Here, it is assumed that a sequential run may be broken if the object is laid out on more than one target.

The average length of a sequential run (in terms of bytes) is described by *run length*. Basically, run length is compared to the stripe unit when deciding on whether a sequential run will be broken across storage targets. Figure 3.8 illustrates a scenario in which sequentiality is disturbed and two targets observe smaller-sized sequential runs. In the figure, the workload W_i has an average request size of 64KB and a run count of 12. As a result, the average run length is 768KB. The stripe unit is taken as 128KB. The sequential run is broken because of striping and each target now observes a sequential run whose average run length is 384KB; it corresponds to a run count value of 6.

The formulas below describe the run count transformation, and Figure 3.9 summarizes the parameters used in the transformation. Except for the stripe unit, other parameters are derived from the workload description parameters (request sizes and run count). The run count parameter does not distinguish between read and write requests; it is the average number of requests of all read and write sequential runs. As a result, in order to obtain the average length of a sequential run in terms of bytes, the read/write mix of the workload is taken into account:

$$\begin{aligned} rp_i &= \frac{\lambda_i^r}{\lambda_i^r + \lambda_i^w} \\ avgRS_i &= rp_i \cdot B_i^r + (1 - rp_i) \cdot B_i^w \\ rl_i &= avgRS_i \cdot Q_i \end{aligned}$$

Parameter		Description
Layout Stripe Unit	(lSU)	The stripe unit that is used while striping an object on more than one target (default: 128KB)
Intermediate (Derived) Parameters		
Read Probability	(rp_i)	The probability of a request being a read request in I/O workload W_i
Average Request Size	($avgRS_i$)	Average request size in bytes in workload W_i
Run Length	(rl_i)	The average size of a sequential run in bytes: $Q_i \cdot avgRS_i$

Figure 3.9: Parameters Used in Run Count Transformation.

Finally, the run count transformation can be formulated as follows,

$$Q_{ij} = \begin{cases} Q_i, & rl_i \leq lSU \\ L_{ij} \cdot Q_i, & rl_i > lSU \cdot M \\ \frac{lSU}{avgRS_i}, & \text{otherwise.} \end{cases}$$

Here, it is assumed that run length is bound by the stripe unit. If the run length of a workload is less than the stripe unit, then the sub-workloads will have the same run count value as the original workload (condition 1). Otherwise, run length is diminished to the stripe unit (condition 3). However, if the run length is too large, then the sequential run will wrap around (condition 2).

Overlap fraction :

$$\Phi_{ij}[k] = \begin{cases} \Phi_i[k], & L_{ij} > 0 \text{ and } L_{kj} > 0 \\ 0, & \text{otherwise.} \end{cases} \quad \forall i, k,\ 1 \leq i, k \leq N \text{ and } \forall j, 1 \leq j \leq M$$

A workload W_i has $N - 1$ overlap fraction parameters, one for each of the remaining objects. It is assumed that, for a given target $target_j$, the sub-workloads (e.g., W_{ij} and W_{kj}) of two objects ($object_i$ and $object_k$) preserve the overlap fractions between them if both objects are laid out on the given target (no matter what L_{ij} and L_{kj} values are). Otherwise, the overlap fractions between these two objects are ignored. This simplification is realistic because when two objects are mapped to a given target, the temporal relation between the I/O requests belonging to these objects is preserved.

Figure 3.10 summarizes the workload description parameter transformations applied by the layout model.

$$
\begin{aligned}
B^r_{ij} &= \begin{cases} B^r_i, & L_{ij} > 0 \\ 0. & \text{otherwise.} \end{cases} \\
B^w_{ij} &= \begin{cases} B^w_i, & L_{ij} > 0 \\ 0. & \text{otherwise.} \end{cases} \\
\lambda^r_{ij} &= \lambda^r_i \cdot L_{ij} \\
\lambda^w_{ij} &= \lambda^w_i \cdot L_{ij} \\
Q_{ij} &= \begin{cases} Q_i, & rl_i \leq lSU \\ L_{ij} \cdot Q_i, & rl_i > lSU \cdot M \\ \frac{lSU}{avgRS_i}. & \text{otherwise.} \end{cases} \\
\Phi_{ij}[k] &= \begin{cases} \Phi_i[k], & L_{ij} > 0 \text{ and } L_{kj} > 0; \\ 0. & \text{otherwise.} \end{cases} \quad \forall i,k,\ 1 \leq i,k \leq N \text{ and } \forall j, 1 \leq j \leq M
\end{aligned}
$$

Figure 3.10: Layout Model for LVM Using Striping

3.2.2.2 Target Model

A target model estimates the utilization of storage target $target_j$ (i.e., μ_j), given the I/O workloads directed at this target (i.e., $W_{ij}, 1 \leq i \leq N$). Most of the existing modeling techniques treat storage targets as black boxes and do not require knowledge of the internal structure of the targets. However, the layout advisor distinguishes two types of targets, and directly applies a black-box technique only for one of the target types. As for the other type, it leverages the knowledge of the internal structure of targets (i.e., a white-box approach). It makes this distinction based on the number of storage devices that a target contains. If a storage target is an individual storage device, a *storage device performance model* is used to estimate the utilization of the device. This performance model treats the storage device as a black-box and reports back the utilization of the device given a set of I/O workload descriptions. Details of this model will be presented shortly. If the target is composed of more than one storage device (i.e., a RAID array) then the layout advisor needs to know how the target is constructed (e.g., RAID level, number of devices, etc.). Therefore, the advisor applies a white-box modeling technique for arrays. Details of the model will be presented shortly. In brief, this model is responsible for generating the I/O workload descriptions for each of the devices in the array. After obtaining the *per-storage device* workload descriptions, *storage device performance models* are employed to estimate the utilization of individual devices. The utilization of the target becomes the maximum of the individual device utilizations.

3.2.2.2.1 Modeling Individual Storage Devices: Storage Device Performance Models A storage device performance model reports the estimated utilization of an individual storage device given a set of I/O workload descriptions.

In theory, any function that maps the I/O workloads to device utilization could be used. However, in addition to device utilization, the layout advisor also needs this model to report the individual contribution of each workload to the device utilization, such that the total device utilization is the sum of those individual loads. As will be explained in Section 3.3.3, the regularization step of the layout algorithm leverages individual workload contributions. Hence, the utilization of the target $target_j$ can be expressed as follows,

$$\mu_j = \sum_{i=1}^{N} \mu_{ij}$$

Here, μ_{ij} represents the load of object $object_i$ on target $target_j$; in other words, the contribution of the workload W_{ij} to the device utilization μ_j. Two request cost models are constructed for a storage device, one model for read requests and the other for write requests. Then, μ_{ij} is estimated as follows,

$$\mu_{ij} = \lambda_{ij}^{r} \cdot \mathcal{COST}^{read}\left(B_{ij}^{r}, Q_{ij}, \chi_{ij}\right) + \lambda_{ij}^{w} \cdot \mathcal{COST}^{write}\left(B_{ij}^{w}, Q_{ij}, \chi_{ij}\right)$$

Here, B_{ij}^{r}, B_{ij}^{w}, λ_{ij}^{r}, λ_{ij}^{w} and Q_{ij} are the parameters of the workload description W_{ij}. $\mathcal{COST}^{read}$ and $\mathcal{COST}^{write}$ are the read and write I/O request cost models of the storage device. The details of the cost model will be explained shortly. Note that each type of device may have a different cost model, and hence different cost functions $\mathcal{COST}^{read}$ and $\mathcal{COST}^{write}$.

The read and write request costs for workload W_{ij} depend on the characteristics of the 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}, $k \neq i$ and $1 \leq k \leq N$, that impinge on the same target. A workload's contribution cannot be determined independently of the other workloads; I/O service time of a workload varies with how the object shares the target it is laid on with other objects. To simplify the task of estimating μ_{ij}, storage device models assume that the impact of all of the workloads (other than ith) for a given valid layout can be captured using a single numeric *contention factor*, χ_{ij}. This factor is essentially an estimate of the number of active workloads competing with the ith workload for the storage device. The contention factor for the ith workload is calculated analytically using the request rates and the overlap fractions of ith workload with the other workloads on the same device. The storage device model then estimates μ_{ij} using only the parameters of W_{ij}, plus the contention factor.

We will give more insight into the *contention factor* concept through two scenarios depicted in Figure 3.11. In Figure 3.11(a), object $object_{ij}$ does not share the target with any other object. As a result, the I/O requests for object $object_i$ that are dispatched to target $target_j$ will not be interfered with by any other requests. In this scenario, the I/O request queue of target $target_j$ can be illustrated as in Figure 3.11(a). The contention factor for the workload W_{ij}, χ_{ij}, is 1; in other

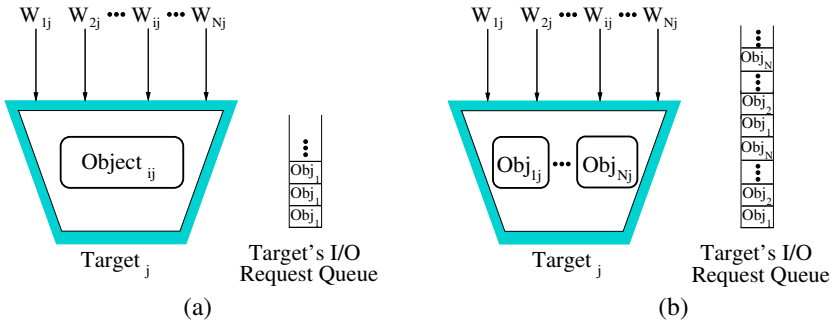


Figure 3.11: Two Scenarios to Illustrate the Concept of *Contention Factor*. (a) The target contains only object $object_i$ and thus I/O requests to the object are not interfered with. (b) The target contains all of the objects and the request rates of all the workloads are the same and all the workloads overlap fully.

words, the number of concurrently running workloads when workload W_{ij} is active is just 1 (only W_{ij} itself). In Figure 3.11(b), target $target_j$ contains all N objects. Assume that the total request rates of all workloads are the same and all the workloads temporally overlap all the time. If it were possible to take the snapshot of the target's request queue, we would see a sequence of I/O requests similar to the one pictured in 3.11(b). There is always a request outstanding for each object, and since all the objects have the same request rate there would be the same number of requests outstanding for each object. As a result, all workloads seem to be actively accessing the target at any time. The contention factor, χ_{ij}, now becomes N because the workload W_{ij} will compete for the disk service along with other $N-1$ workloads. Here, some of the complexity is eliminated by taking into consideration only the request rate of other competing workloads. It is assumed that run counts and request sizes of the competing workloads do not play an important role in the estimation of contention factor. In conclusion, the layout advisor estimates the contention factor for a given workload W_i as follows:

$$\chi_i = \sum_{k=1}^{N} \Phi_i[k] \cdot \frac{\lambda_k^r + \lambda_k^w}{\lambda_i^r + \lambda_i^w}.$$

With this simplification, the per-request costs $\mathcal{COST}^{read}()$ and $\mathcal{COST}^{write}()$ are functions of the properties of storage device and three workload parameters: request size, run count (sequentiality), and the contention factor.

Here, both cost functions are black-box cost models for a given storage device. It is possible, but difficult, to build accurate analytic cost models [30, 34, 52, 53]. Instead, black-box models are constructed based on interpolation among tabulated measurements of storage device performance; that is, two three-dimensional look-up tables are constructed (one for read and another for write requests). There are many types of storage devices in the market, with diverse characteristics. Therefore, it would be a relatively hard task to design an analytical cost model for each of

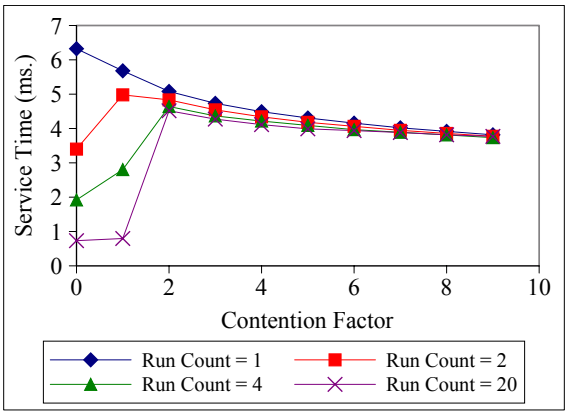


Figure 3.12: Service Time (Cost) Model of SEAGATE ST318453LC Hard Disk for 8 KByte Read Requests

them. However, using a black-box technique, any type of storage device (from rotational to solid-state devices) can be modeled using the same technique with the same amount of effort.

Others have used similar techniques to build black box models for disk arrays and storage devices [6, 55]. The models are constructed by subjecting the storage devices 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 $\mathcal{COST}^{read}()$ or $\mathcal{COST}^{write}()$ for a workload, the layout advisor looks up the tabulated cost from the appropriate table, interpolating among nearby calibration points if necessary. Appendix C gives detailed information about how a look up table is constructed for a given storage device, and how the interpolation is conducted.

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 3.12 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 quickly and 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.

Parameter	Description
RAID level ($RAIDx_j$)	RAID level (or configuration) used to construct the array.
Number of Devices (D_j)	Number of devices in the array.
Stripe Unit (SU_j)	Stripe unit of the RAID array.
Storage Device Types ($T_1 \dots T_{D_j}$)	A target contains D_j many devices. Each device may have a distinct type.

Figure 3.13: Parameters that Define Target $target_j$.

3.2.2.2.2 Performance Modeling of RAID Arrays In the preceding section, we have presented how the layout advisor models storage targets which are individual storage devices. Here, we will discuss how it estimates the utilization of a target which is composed of more than one storage device. It is assumed that targets containing multiple storage devices are constructed using a RAID configuration. RAID arrays differ from each other with respect to the attributes presented in Figure 3.13. The layout advisor models RAID arrays using a white-box technique, and needs to know those attributes for all of the storage targets in the system. Therefore, in addition to targets' storage capacities c_j (see Figure 3.3), those attributes (per-target) are also given to the layout advisor as input parameters.

Figure 3.14(a) is the illustration of the data placement problem, the same figure was presented in Section 3.1.2 (see Figure 3.2). It depicts the problem of finding out a layout for each object on a subset of the available targets. Figure 3.14(b) depicts the situation after the layout model transforms workload descriptions based on a particular layout. Note that Figure 3.14(b) can be viewed as a special and trivial instance of Figure 3.14(a) because the layout is *not* unknown anymore. As a result, similar to what layout model does, the layout advisor can transform I/O workload descriptions (directed at a given target) to generate the set of I/O workload descriptions for each storage device in the target. Since a target with multiple devices is a RAID array, the transformations taking place within the target will be called *RAID transformations*.

After obtaining the I/O workload descriptions for each storage device, storage device performance models (presented in the preceding section) can be employed to estimate the utilizations of individual devices in the array. Hence, the problem of finding the utilization of a target can be diminished into simpler sub-problems: finding the utilizations of the devices within the target. Finally, the utilization of target $target_j$ becomes the utilization of the most utilized device:

$$\mu_j = \max_{k=1}^{D_j} \mu_j^k$$

Here, μ_j^k represents the utilization of the kth device in target $target_j$. What follows is a more detailed explanation of how workload descriptions are transformed to generate per-device workloads.

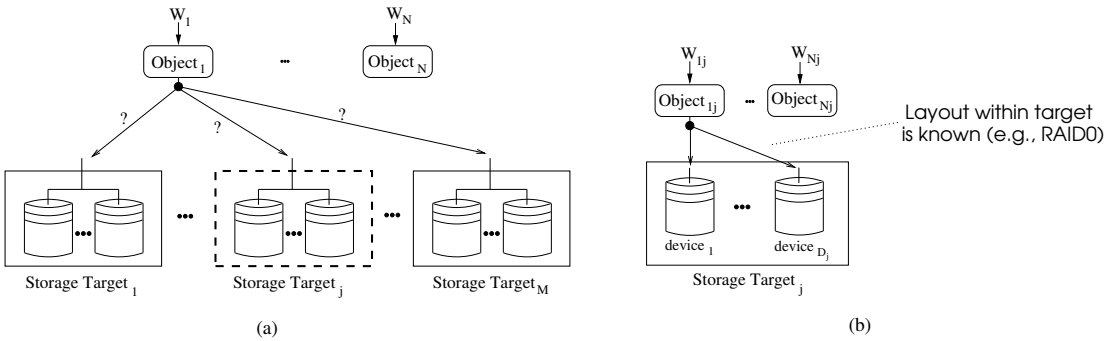


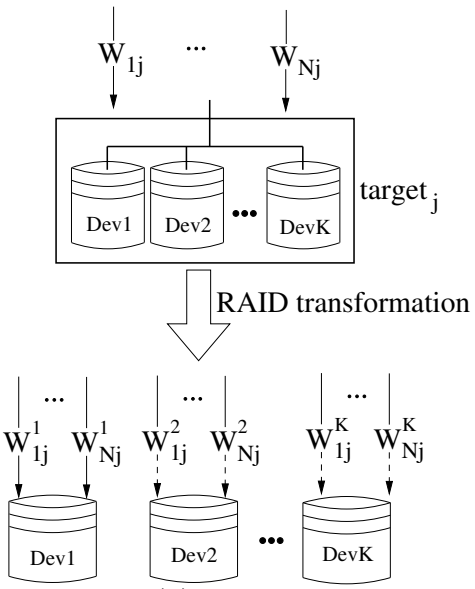
Figure 3.14: White-box approach to RAID array modeling can be viewed as a special case or instance of the data placement problem.

Figure 3.15(a) illustrates the situation *after* the layout model transforms workload descriptions. Since it is assumed that the performance of a target does not depend on another, each target's utilization can be estimated independently. Like a layout model, a RAID transformation model further transforms I/O workload descriptions to obtain workload descriptions per storage device. Basically, each I/O workload description directed at target $target_j$ ($W_{ij}, 1 \leq i \leq N$) is transformed to obtain per-storage device I/O workloads ($W_{ij}^d, 1 \leq i \leq N$ and $1 \leq d \leq D_j$). What RAID transformation does is simply to reflect the behaviour of the given RAID array onto each workload. For example, RAID0 transformation applies almost the same set of transformations as the one applied by the layout model. In RAID0 transformation, unlike the layout model, there is no default stripe unit. Instead, each target has its own stripe unit (SU_j) and all objects are laid out on all devices evenly; that is, the layout is fixed and known. As a result, RAID0 transformation can be viewed as a special instance of the layout model. Figure 3.15(b) summarizes how workload description parameters are transformed for a RAID0 target.

For other RAID levels, similar set of transformations can be applied. For example, RAID1 transformation applies exactly the same transformations as RAID0. However, since each storage device is a mirror of another, the number of devices in a target is taken as $\frac{D}{2}$. In case of RAID4 arrays, write requests of a workload incur additional read workload if the workload's write request size is less than stripe size of the array (i.e., $SU \cdot D$). Moreover, RAID4 transformation adds a new workload description for the parity device(s). RAID5 transformation is similar to RAID4 transformation, except there is no need to generate a separate parity workload.

3.2.3 Existing Approaches to Storage System Modeling

We have introduced the storage system performance model adopted by the layout advisor. In this section, we will present some other existing approaches that are used to model storage systems or storage targets, and show that the proposed technique can be considered as a hybrid.



(a) Illustration

$$
\begin{aligned}
B^r_{i(k,j)} &= B^r_{ij} \\
B^w_{i(k,j)} &= B^w_{ij} \\
\lambda^r_{i(k,j)} &= \lambda^r_{ij} \cdot \frac{1}{D_j} \\
\lambda^w_{i(k,j)} &= \lambda^w_{ij} \cdot \frac{1}{D_j} \\
Q_{i(k,j)} &= \begin{cases} Q_{ij}, & Q_{ij} \cdot avgRS_i \leq SU_j \\ \frac{1}{D_j} \cdot Q_{ij}, & Q_{ij} \cdot avgRS_i > SU_j \cdot D_j \\ \frac{SU_j}{avgRS_i}, & \text{otherwise.} \end{cases} \\
\Phi_{i(k,j)}[m] &= \Phi_{ij}[m],\ \forall i, m, 1 \leq i, m \leq N.
\end{aligned}
$$

(b) RAID0 Transformations

Figure 3.15: RAID Transformations. (a) Each workload description is transformed based on the configuration of the target: RAID level, stripe unit and number of devices. (b) RAID0 Transformations. I/O workload description of object $object_i$ is transformed to generate the workload description impinged on the kth device in RAID0 target $target_j$.

There are two main approaches to system modeling, based on whether the internals of the system to be modeled are known or not: black-box models and white-box models. Black-box modeling requires no knowledge of the internals of the storage system being modeled, while white-box modeling requires deep understanding of the internal implementation of the system. In a black-box approach, the system's performance is measured using a set of controlled workloads, and performance measurements are collected. Those measurements are used either to design an analytical model through regression or to construct a look-up table. In a white-box approach, a system can be modeled analytically using a mathematical model since *how* the system works is exposed to the model designer.

System modeling approaches can further be classified based on *how* they *implement* the model: analytical models and tabular models. As described above, a black-box or a white-box approach uses either an analytical model or a tabular model to describe the system. Both analytical and tabular models (e.g., [6, 52, 53, 55, 30]) are preferred over trace-driven simulations (e.g., [59, 11]) because they are much faster. Considering that the storage system performance model will be employed frequently to compare a large set of valid layouts to identify good ones, the efficiency of the model plays a very critical role.

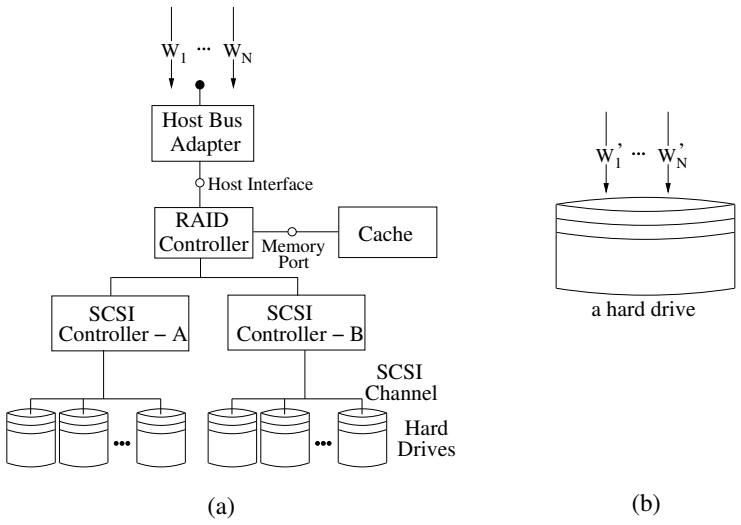


Figure 3.16: A Modular View of a Storage System. Delphi models each component of a storage system. A workload is transformed by each component until its impact on each hard drive is estimated.

3.2.3.1 Analytical Models

Delphi [52], developed at HP Laboratories, offers a modular analytical model which reports utilizations of individual devices (i.e., hard drives). Figure 3.16(a) depicts a typical storage system that can be modeled by Delphi. Delphi is a modular model because it models the components constituting a storage system separately, and one model's output is taken as an input by the other. In brief, I/O workloads (represented using the Rome model) are transformed by each component until workloads per-hard drive are obtained as depicted in Figure 3.16(b). This is very similar to what the proposed layout and target (RAID transformations) models do. Delphi and the proposed approach differ after obtaining per-storage device I/O workload descriptions.

Delphi, as a fully analytical model, models the performance of a hard drive analytically, too. A drive's performance is characterized by its sustained read/write transfer rates and average seek time, and the behavior of the drive is mathematically modeled to estimate the device utilization for a given set of workloads. However, it is not trivial to mathematically model the behavior of a storage device correctly, and doing so requires deep understanding of the internal implementation of the device. As discussed in the preceding section, performance of a storage device is highly non-linear. In addition, modern storage devices employ complex algorithms whose details are not exposed to their users. Therefore, analytically modeling a device's performance can potentially be inaccurate.

Curve-fitting models (or regression) can be used in place of the modular model described above, or as a complement to it. Curve-fitting is not generally employed

to model a complete storage system but rather individual devices or arrays. To build the model, a storage target is subjected to calibration workloads with known workload parameters, and the performance (e.g., throughput or average response time) is measured for various parameter values. Those measurement points are used to design some form of function (from workload parameters to performance) which is fitted to these measurements. For example, Varki et al. suggested such a performance model for disk arrays [53]. They parametrize the I/O workload with a set of parameters such as request size, run count, and the number of concurrent workloads. They also take some of the device characteristics like array cache size, sustained transfer rate, average read/write seek time, stripe unit, number of disks in the array as inputs to their performance model. Curve fitting techniques are used to model certain performance metrics; for example, disk seek time as functions of these parameters.

Since the performance of a storage device can be highly non-linear, analytical models which perform curve-fitting are potentially more accurate than models like Delphi. However, in general, white-box techniques can be used to build a single parametrized model covering a whole class of devices. As a result, it is flexible where there is a need to model a large set of storage targets. On the other hand, to generate a model using curve-fitting technique for a new storage target, the whole cycle of obtaining performance measurements using controlled workloads and then curve fitting has to be performed again.

3.2.3.2 Table-based Models

Table-based models have similarities to curve fitting models. First of all, they are typically not used to model a whole storage system but rather individual storage devices or arrays. Secondly, a synthetic workload generator is used to generate calibration workloads. However, in table-based modeling, instead of doing regression, a look-up table is constructed using the performance measurements taken at different workload parameter settings.

Anderson [6] presents a table-based model for a disk array. The I/O workload is parametrized by using request type, request size, sequentiality, and average request queue length. For each of those parameters a set of values are determined, and for each possible combination of values an I/O workload is generated and run against the disk array. For each workload, the performance of the array is measured and the look-up table is populated. The layout advisor uses similar look-up tables to estimate the utilizations of *individual devices* rather than disk arrays.

Wang et al. propose to improve table-based models by employing machine learning tools [55]. In brief, they use classification and regression tree (CART) modeling, which can be viewed as a type of non-linear regression tool. CART modeling is used to approximate functions on a multi-dimensional Cartesian space using piecewise-constant functions. As a result, their method can also be viewed as a curve-fitting model. It is useful when the number of workload parameters and thus number of possible calibration points increase.

3.2.3.3 Layout Advisor's Storage System Performance Model

The storage system performance model adopted by the layout advisor is a hybrid of modular analytical modeling and the tabular method. The layout model and the RAID transformations performed within each multi-device target resemble Delphi's modular analytical model. Storage device performance models to estimate individual device utilizations use tabulation in order to model the performance of a storage device. Modeling the storage system infrastructure analytically makes the proposed system model flexible. A database or storage administrator can easily generate what-if scenarios by designing various storage targets. Modeling the performance of individual devices using tabulation increases the accuracy of the performance estimations.

3.3 Recommending Workload-Aware Optimized Layouts

The two key elements of the proposed algorithm 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. Initially, some ad hoc heuristics were explored for finding optimized layouts, but ultimately we found the generic solvers to be fast and effective. By using a generic solver, the solver's techniques for exploring the optimization space can directly be leveraged. Thus, instead of re-inventing the wheel or designing heuristic algorithms, the state of art techniques incorporated into NLP solvers can be leveraged. Explicitly formulating layout as a non-linear programming problem also 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, such constraints can easily be added to the NLP problem before solving it.

3.3.1 Solver

We use AMPL [23], a C-like standard mathematical modeling language, to formulate the layout problem as a non-convex NLP. There exist several general solvers designed to solve non-convex optimization problems with non-linear objective functions and non-linear constraints. We use the MINOS solver [36], because it supports the use of external (i.e., non-AMPL), black-box functions as part of the definition of the objective function. This feature is leveraged for the storage device performance models (black-box tabular models).

In order to use a generic NLP solver like MINOS, several issues must be addressed. First, a valid initial layout must be provided. This step is described in more detail in Section 3.3.2. Second, the solver will find a valid, optimized layout, but the resulting layout may not be regular. If a regular layout is required, then

```
1  solution = EMPTY;
2  for(i = 1; i <= NoOfIterations; ++i){
3
4      init_layout    = FindAValidInitialLayout();
5      general_layout = CallAnNLPSolver(init_layout);
6
7      if(LookingForaRegularizedSolution)
8          new_solution = Regularize(general_layout);
9      else
10         new_solution = general_layout;
11
12     if(solution == EMPTY || BetterThan(new_solution, solution))
13         solution = new_solution;
14 }
```

Figure 3.17: Layout Algorithm. Here, the function FindAValidInitialLayout() is assumed to generate a different initial layout each time it is invoked.

the solver's solution has to be *regularized*. This regularization step is described in Section 3.3.3. Figure 3.17 summarizes the behavior 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 3.17 incorporates this iteration (i.e., the outer-most *for* loop). 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 the layout advisor. All of the results presented in Experimental Evaluation section use a single initial layout (i.e., $NoOfIterations$=1). The choice of a suitable initial layout is detailed below.

3.3.2 Initial Layout

Originally, SEE was used 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, the layout advisor uses 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, as described in the preceding section.

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. In addition, if there are tight space constraints, it is possible that this algorithm will fail to provide a layout (even if such a layout is possible) and manual intervention would be necessary.

3.3.3 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, a solver that is intended for combinatorial problems should be used. Thus, instead, the solver is used 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 $object_i$, $\sum_{j=1}^{M} \mu_{ij}$) 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 of that object. 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 $(33\%, 33\%, 33\%)$.[2] 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 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. These are called *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.

3.3.4 Putting Things Together

Here, we will present how different stages of the layout optimization algorithm are glued together by AMPL. The heart of the optimization algorithm is the storage system performance model described in Section 3.2.2. To recap, this model accepts I/O workload information for a set of database objects and the description of a set of storage targets, and reports back *the maximum target utilization* for a given valid layout. The NLP solver and the regularizer use this metric to asses the quality of valid layouts. Figure 3.18 depicts how the AMPL-coded storage system model sits at the heart of the layout optimization process. More details on the AMPL-coded model can be found in Appendix D.

3.4 Experimental Evaluation

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

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

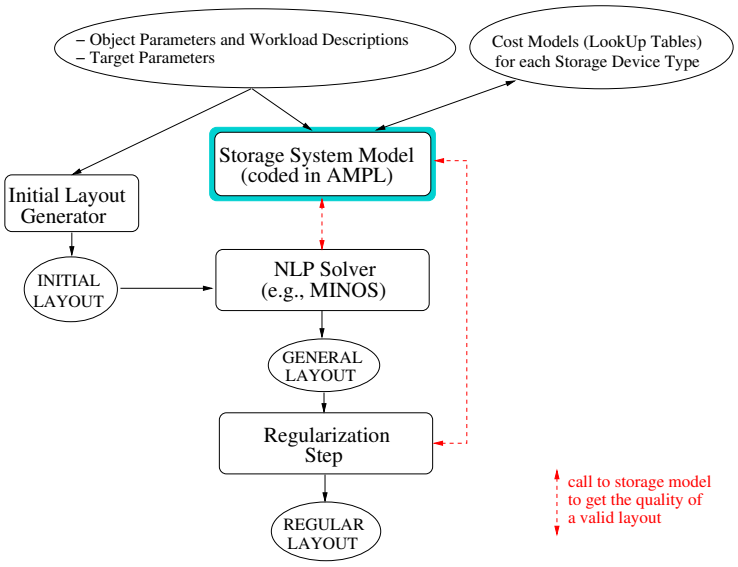


Figure 3.18: Putting the Steps of Layout Optimization Together. AMPL-coded storage system model is called by the NLP solver and the regularizer to assess the quality of valid layouts.

3.4.1 Experimental Setup

The storage devices we have considered for the evaluation consist of rotational hard drives and a solid-state drive, and the objects to be laid out are individual database tables, indexes, logs, and a tablespace for temporary objects. 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. In most cases, the 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 the 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 a layout. These estimated utilizations are used to illustrate and explain the advisor's behavior. Although target utilizations are directly measured in the experimental environment, we cannot compare utilizations measured under the optimized layout with utilizations measured under the baseline layouts. This is because the experimental setup has a closed loop testing environment. Optimized layouts make queries run faster and hence cause new queries to be submitted more quickly. Thus, the optimized layouts tend to have higher measured utilizations than the baseline layouts simply because they have higher query throughput. However, we can directly compare the estimated utilizations for the baseline and optimized layouts, because both are computed using the query rates and other workload characteristics specified in the input workloads (the W_i's). Any improvements in the estimated utilizations are the result of improvements in the quality of the layout.

Database	Total Size	Number of Objects			
		Tables	Indexes	Temporary Space	Transaction Log
TPC-H	9.6GB	8	11	1	0
TPC-C	9.2GB	9	10	0	1

Figure 3.19: Databases Used in the Experiments

In the experiments, we have used the PostgreSQL database management system, version 8.0.6. We experimented with two databases: a scale factor 5 TPC-H database and a scale factor 90 (i.e., number of warehouses) TPC-C database. We used the open source implementation of the TPC-H and TPC-C benchmarks [40]. The numbers of objects in these databases and their total sizes are shown in Figure 3.19. Note that the total sizes do not include the sizes of the transaction log and temporary tablespace. In an active system, those two objects may vary in size depending on several factors such as the sizes of other objects, number of queries concurrently running in the system, read/write mix of the workload, available system memory, the writeback settings of the operating system.

We define four SQL workloads running against these databases, as shown in Figure 3.20. Three of those SQL workloads are online analytical processing (OLAP) workloads. OLAP workloads run against the TPC-H database. Normally, the TPC-H benchmark defines 22 query types; however, Query 9 is not included in our workloads because of its excessive run-time compared to the run times of the other queries. While the average run time of other queries is around 10 minutes, the run time of Query 9 is around 260 minutes in our system. The `OLAP1-63` workload consists of 63 TPC-H queries, each TPC-H query type occurs three times in the mix. The entire mix is randomly permuted and the queries are executed sequentially with no think times. `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 last OLAP workload is `OLAP1-21`. The `OLAP1-21` workload consists of 21 queries, each TPC-H query type occurs once in the mix. The queries are executed sequentially in a randomly selected order, with no think times. Finally, the `OLTP8-90` workload generates an online transaction processing (OLTP) workload running against the TPC-C database. The `OLTP8-90` workload is generated by eight simulated terminals per warehouse, one database connection per terminal and with no think time or keying time.

The PostgreSQL database management system is deployed 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 were used to build the workload models, as described in Section 3.2.1. The size of the database system's shared buffer is set to the maximum allowable value, 2GB, for all of the OLAP workloads (`OLAP1-63`, `OLAP8-63` and `OLAP1-21`), and to 1.5GB for the OLTP workload

Workload Type	SQL Workload	Concurrency Level	Number of Queries	Target Database
OLAP	OLAP1-63	1	63	TPC-H
	OLAP8-63	8	63	
	OLAP1-21	1	21	
OLTP	OLTP8-90	8x90	n/a	TPC-C

Figure 3.20: Query Workloads Used in the Experiments

Config. Name	No of Targets	Type of Mixture	No of Devices in Targets	Explanation
4RAID0	1	n/a	{4}	Single target which is a 4-disk RAID0 array.
4JBOD	4	Homog.	{1,1,1,1}	Four identical targets, each is a standalone disk.
2HETERO	2	Heterog.	{3,1}	Two targets: a target is a 3-disk RAID0 array, the other is a standalone disk.
3HETERO	3	Heterog.	{2,1,1}	Three targets: a 2-disk RAID0 array and two standalone disks.
4JBOD+SSD	5	Heterog.	{1SSD} + {1, 1, 1, 1}	Five targets: a SSD and four standalone disks.

Figure 3.21: Storage Target Configurations Used in the Experiments

(OLTP8-90). 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 (SEAGATE ST318453LC [51]) behind a configurable Dell Perc 4Di RAID controller, and a 32GB solid-state drive (Patriot PE32GS25SSDR [33]) behind a 3Gb/s Koutech SATA-II controller. The performance of those two types of storage devices under various workloads is presented in Appendix C (see Figure C.4). These five storage devices are used to hold the TPC-H and TPC-C database objects. Database objects will be laid out on the devices using different configurations. Figure 3.21 summarizes those configurations that were used in the experiments. I/O workload information for all of the SQL workloads is obtained by running those workloads under the 4RAID0 target configuration.

3.4.2 Layout Quality: Homogeneous Systems

In this set of experiments, we compare DBMS performance under advisor-recommended layouts with the performance under a baseline stripe-everything-everywhere (SEE) layout. We have used the OLAP1-63 and OLAP8-63 workloads, and the 4JBOD tar-

Workload	Execution Time (seconds)		Speedup (Gain)
	Baseline (SEE)	Optimized Layout	
OLAP1-63	40927	31879	1.28x
OLAP8-63	16201	13527	1.2x

Figure 3.22: Execution Times of the OLAP1-63 and OLAP8-63 Workloads Using Baseline (SEE) and Optimized Layouts under the 4JBOD Configuration.

get configuration. For each workload, the layout advisor is asked to recommend a layout of the TPC-H database objects onto the four identical targets. As discussed in Section 3.1.1, SEE is the most widely advised layout technique by database vendors, especially in homogeneous systems. Therefore, SEE is used as the baseline layout.

Figure 3.22 summarizes the results. It presents the total execution time of the SQL workloads when the baseline and advisor recommended layouts were used. The last column of the figure shows the speed-up obtained when the optimized layouts were used. Although this scenario is well-suited to SEE, the layout advisor is able to obtain some performance improvement for both workloads. Figure 3.23(a) illustrates the optimized regular layouts that the advisor recommends for the OLAP1-63 and OLAP8-63 workloads under the 4JBOD target configuration. Objects are shown in decreasing order of request rate, and the layout of only first ten objects is showed. For both OLAP1-63 and OLAP8-63, the recommended layouts separate the two most heavily-requested objects (LINEITEM and ORDERS). For OLAP8-63, the workload on LINEITEM is less sequential than it is under OLAP1-63, because of concurrency. As a result, the performance penalty for interference with LINEITEM is lower under the OLAP8-63 workload, and thus LINEITEM is not completely isolated in the OLAP8-63 layout. Instead, the layout advisor distributes I_L_ORDERKEY and TEMP SPACE across all of the targets to better balance the load.

Figure 3.24 illustrates the behavior of the layout advisor by showing the quality of the layouts that it considers at different stages of its execution. In the figures, each group of bars shows the estimated utilization (μ_j) of one of the four disks. The leftmost bar in each group shows the advisor's estimated utilization for the baseline SEE layout, for the purpose of comparison. The second bar in each group shows the estimated target utilization under the *initial* layout generated by the advisor as a starting point for the NLP solver. These initial layouts assign each object to a single target. They tend to be unbalanced, as is the case for both OLAP1-63 and OLAP8-63, because the advisor does not take interference, workload sequentiality, and the other factors into account, but rather employs a simplistic algorithm when generating them. In addition, the resulting layout minimizes the I/O parallelism because each object is mapped to a single target.

The third bar in each group shows the estimated utilization for the (non-regular)

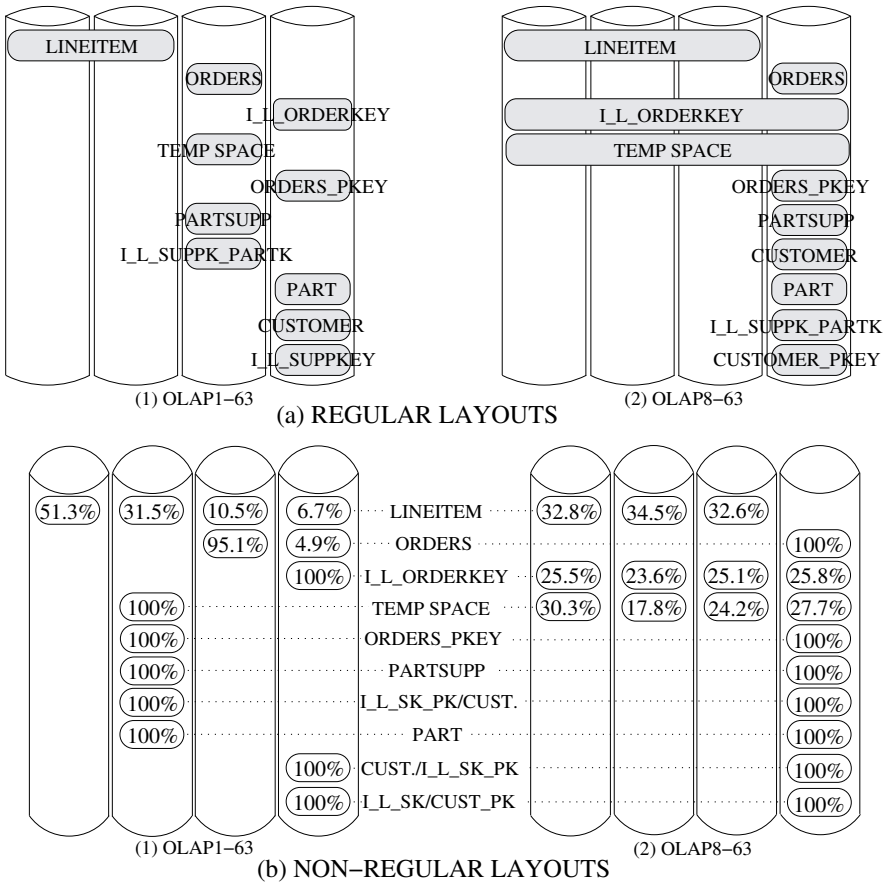
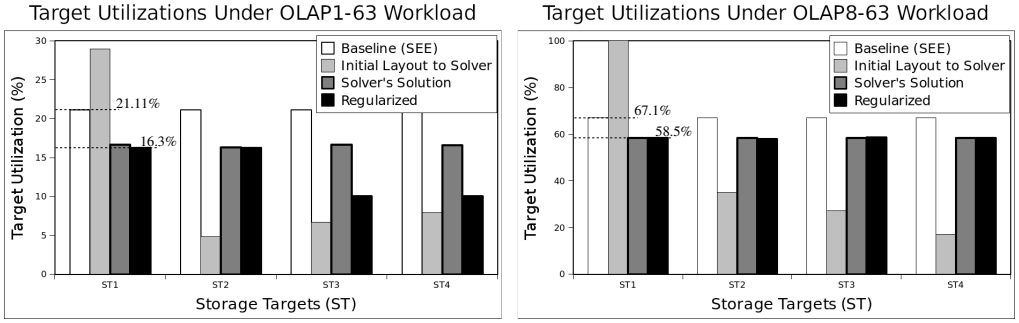


Figure 3.23: Optimized (a) Regular and (b) Non-regular (General) layouts of the TPC-H Objects for the `OLAP1-63` and `OLAP8-63` Workloads under `4JBOD`.



(a) `OLAP1-63` (b) `OLAP8-63`

Figure 3.24: Estimated Target Utilizations for the `OLAP1-63` and `OLAP8-63` Workloads under `4JBOD`.

layout identified by the NLP solver. The solver's layouts for both OLAP1-63 and OLAP8-63 are shown in Figure 3.23(b). For both workloads, this layout is very balanced, and it reduces *the maximum target utilization* relative to the default SEE layout, which is also balanced. These reductions are achieved by reducing interference among the objects and at the same time by mapping some objects to multiple targets to distribute the load. Thus, the layout advisor could address this trade-off effectively for the given I/O workloads and target configuration.

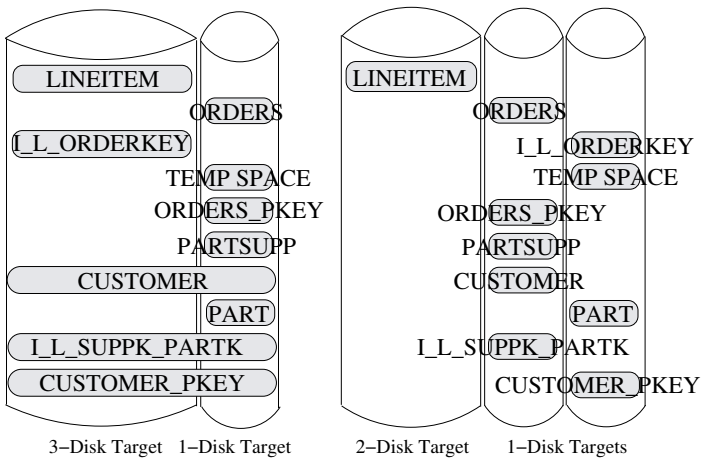
If, in the host system, the layout mechanism supports non-regular layouts, then the layouts shown in Figure 3.23(b) can be implemented directly. If not, then the final regularization step is needed. The fourth bar in each group in Figure 3.24 shows the estimated utilizations under the final regularized 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. In the case of the OLAP8-63 workload, the solver's layout (Figure 3.23(b)(2)) is almost regular, and the resulting regularized layout (Figure 3.23(a)(2)) is very close to the solver's. For the OLAP1-63 workload, the solver's layout (Figure 3.23(b)(1)) is less regular, and regularizing the layout of the heavily-requested LINEITEM table creates some load imbalance, which the regularizer is unable to correct completely. Nonetheless, the maximum target utilization under the regularized layout is almost the same as it is under the solver's layout.

3.4.3 Layout Quality: Heterogeneous Systems

In this set of experiments, the layout advisor is asked to recommend layouts given heterogeneous systems. For these experiments, we have used the OLAP8-63 workload, and 2HETERO, 3HETERO and 4JBOD+SSD target configurations. 2HETERO is composed of two targets: a RAID0 array of 3 disks and an individual disk. 3HETERO contains three targets: a RAID0 array of 2 disks and 2 individual disks. 4JBOD+SSD is a mix of 4 individual disks and the SSD. The server's RAID controller was used to create 2HETERO and 3HETERO configurations out of 4 identical rotational hard drives. Therefore, both systems have been configured to be heterogeneous. We will first analyze the optimized layouts for this kind of heterogeneous systems. In practice, heterogeneity may arise not only from the storage system configuration, but also from the presence of multiple types of devices; 4JBOD+SSD configuration represents such a scenario. The fifth device, which is a solid state drive, has different performance characteristics than the rotational hard drives. As a result, the system itself is heterogeneous by nature. In the second part, we will analyze the layout advisor's behavior for this kind of heterogeneous systems.

3.4.3.1 Configuration Heterogeneity

Unlike the previous scenario, in which a homogeneous system is considered, in heterogeneous systems it is not obvious how to define baseline layouts against which



(a) 2HETERO configuration (b) 3HETERO configuration

Figure 3.25: Optimized Layouts of the TPC-H objects for the OLAP8-63 Workload under the 2HETERO and 3HETERO Configurations.

Storage Target Config.	Baseline (SEE) Execution Time	Baseline (isolate tables) Execution Time	Baseline (isolate tables, indicies & temp) Execution Time	Optimized Execution Time	Speedup (Optimized vs. SEE)
2HETERO	18715	14507	n/a	13317	1.41x
3HETERO	16922	n/a	22359	13163	1.29x
4JBOD	16201	n/a	n/a	13527	1.2x

Figure 3.26: Workload Execution Times (in seconds) of the OLAP8-63 Workload using the Baseline and Optimized Layouts under the 2HETERO and 3HETERO Heterogeneous Target Configurations.

to compare the layout advisor's recommendations. As the first baseline SEE is used, although it is clear that this will lead to load imbalance when the targets are heterogeneous. For the 2HETERO configuration, we consider 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 3HETERO 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 the layouts that might be considered by a database administrator faced with these situations. Figures 3.25(a) and 3.25(b) show the layouts recommended by the advisor for the 10 most heavily-requested objects for the 2HETERO and 3HETERO configurations, respectively.

Figure 3.26 summarizes the results of these experiments. In addition to the results for the heterogeneous 2HETERO and 3HETERO target configurations, we have also included the results from Figure 3.22 for the same workload (OLAP8-63) under

the homogeneous 4JBOD 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 3HETERO than under 4JBOD, and worse under 2HETERO than under 3HETERO. Specifically, SEE gives about the same performance under 3HETERO as it did in the homogeneous configuration, but it starts to suffer in the 2HETERO configuration because of load imbalance. In 2HETERO, both targets receive the same amount of workload. Thus, the large target is underutilized and the system's performance is bounded by the small target. The alternative baseline layouts improved on SEE in the 2HETERO case, but the layout advisor provided a larger improvement. In the 3HETERO configuration, the alternative baseline actually performed worse than both SEE and the optimized layout. Isolating tables, indexes and temporary tablespace in the 3HETERO configuration hurt the performance significantly relative to the SEE baseline, which points to the difficulty of using heuristic layout guidelines.

3.4.3.2 Device Heterogeneity

Next, we will analyze the DBMS performance for OLAP8-63 workload under the 4JBOD+SSD target configuration. As the first baseline layout, SEE is used. We consider a second baseline layout that places all objects on the solid-state drive. Since OLAP workloads are mostly read-only and the solid-state drive has superior read I/O performance than the rotational hard drives (see Figure C.4) and has enough space to store all objects, this layout could be considered by a database administrator. We also consider some scenarios in which the capacity of the SSD drive is reduced. The capacity of the SSD drive is 32GB and the capacity of each rotational hard drive is 18GB. The total size of the 19 TPC-H objects is around 10GB[3], and the temporary tablespace for OLAP8-63 workload can grow up to 10GB. Other configurations are obtained by restricting the capacity of the SSD to 10GB, 6GB and 4GB. In all of those three cases, placing all objects on the SSD is not possible. In such mixed systems, it is indeed common that the SSD is not large enough to hold everything as it is more expensive than rotational drives. Figure 3.27 depicts the advisor recommended layouts of the 10 most heavily-requested TPC-H objects for all of the four cases.

Figure 3.28 summarizes the results of the experiments. The most important observation is that the layout advisor was able to identify a good layout in all cases. 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

[3]The four largest objects are LINEITEM, ORDERS, PARTSUPP and I_L_SUPPKEY_PARTKEY with sizes 4.98GB, 1.12GB, 0.68GB and 0.62GB, respectively. There are three indexes defined on LINEITEM with sizes 0.5GB. The sizes of the rest of the objects range from 190MB to 8KB with an average size of 75MB.

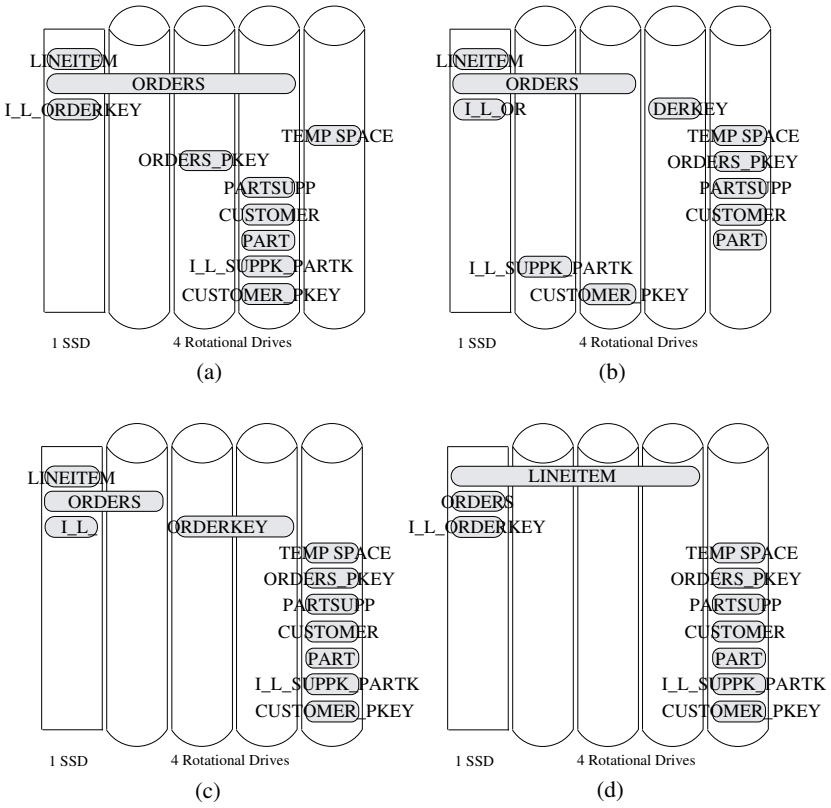


Figure 3.27: Optimized Layouts for the `OLAP8-63` Workload under `4JBOD+SSD` Configuration. The capacity of the SSD is taken as (a) 32GB, (b) 10GB, (c) 6GB, and (d) 4GB.

was sufficient to allow it, placing all of the objects on the SSD resulted in much better performance, but this layout fails to utilize the disk drives at all. However, the advisor recommended layout led to slightly better performance. 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, in all of the other three cases in which SSD does not have enough capacity to store all objects, the layout advisor could still recommend good layouts. For example, even with only a 6GB SSD, the optimized layout still performs better than the SSD-only layout with a 32GB SSD. As a result, the layout advisor could leverage the SSD and found a non-trivial good mapping of the objects by exploiting the knowledge on the I/O workload and the targets' performance characteristics.

It is also instructive to compare execution times from the SSD experiments (Figure 3.28) to those from the disk-only experiments (Figure 3.26). For example, the workload runs in 16201 seconds under SEE in the four disk `4JBOD` configuration (with no SSD), or in 13527 seconds in the same configuration with an optimized layout (Figure 3.26). With the addition of a 4GB SSD to the four disks, the recommended layout allows the workload to finish in 8529 seconds (Figure 3.28)-

Capacity of the SSD	Baseline (SEE) Execution Time	Baseline (all objects on SSD) Execution Time	Optimized Execution Time	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 3.28: Workload Execution Times (in seconds) for the Baseline and Optimized Layouts for the `OLAP8-63` Workload under the `4JBOD+SSD` Target Configuration.

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.

Lastly, comparing the layouts of `orders` table and `i_l_orderkey` index in the SSD experiments to those in the disk-only experiments reveals that the layout advisor does a good job in matching objects' I/O characteristics and devices' different performance characteristics. In `OLAP8-63`, `orders` table is sequentially accessed and `i_l_orderkey` index is almost randomly accessed, and both objects are highly co-accessed. In disk based systems, having these objects share the same target would prevent the target from exploiting the sequentiality so the layout advisor tend to separate these objects (Figures 3.23(a)(2) and 3.25). However, an SSD-based target is indifferent to contention for read requests (see Figure C.4) so both objects can share the same SSD target (Figure 3.27(d)).

3.4.4 Layout Quality: Consolidation Scenario

Here, we consider a consolidation scenario in which two database system instances run on the server. One instance handles the `OLAP1-21` workload and the other `OLTP8-90` workload. In this scenario, there are a total of 40 database objects to be laid out, including 20 from the TPC-H database and 20 from the TPC-C database. Two target configurations are considered: `4JBOD` and `4JBOD+SSD`.

3.4.4.1 Homogeneous System

The layout advisor is asked to generate an optimized layout of the 40 objects under the `4JBOD` target configuration, and we compare this layout against a SEE baseline, which stripes all 40 objects across the four storage targets. In this experiment, the performance of the `OLAP1-21` workload is measured in terms of the wall-clock time required to complete the SQL workload. For the `OLTP8-90` workload, performance

Workload	SEE Baseline Performance	Optimized Performance	Improvement Optimized vs. SEE
OLAP1-21	24416 sec.	17005 sec.	1.43x
OLTP8-90	304 tpmC	360 tpmC	1.18x

Figure 3.29: Consolidation Scenario Performance Under the Baseline and Optimized Layouts for the 4JBOD Target Configuration.

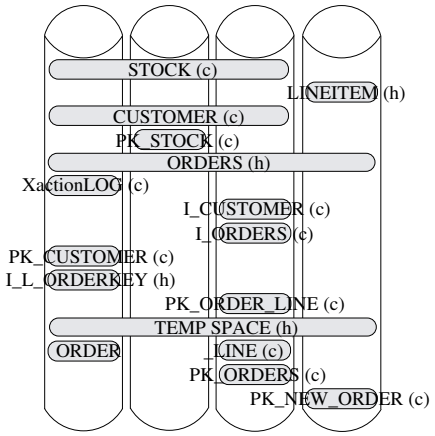


Figure 3.30: Optimized Layouts of the TPC-H (h) and TPC-C (c) Objects for the Consolidation Scenario.

is measured in TPC-C New-Order transactions per minute (tpmC). The OLTP8-90 workload runs 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 1400 seconds.

Figure 3.29 summarizes the results of this experiment, and Figure 3.30 shows the regular layout recommended by the advisor for the 15 most heavily-requested objects. Optimization boosts the performance of both the OLAP1-21 and OLTP8-90 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.

3.4.4.2 Heterogeneous System

Next, the layout advisor is asked to find an optimized layout for the same workload under 4JBOD+SSD configuration. SEE is used as the initial baseline, and the second baseline layout places all objects on the SSD. As was noted in the preceding section, the total size of the 19 TPC-H objects is around 10GB, and the temporary tablespace can grow up to 5.5GB for the OLAP1-21 workload. The total size of the

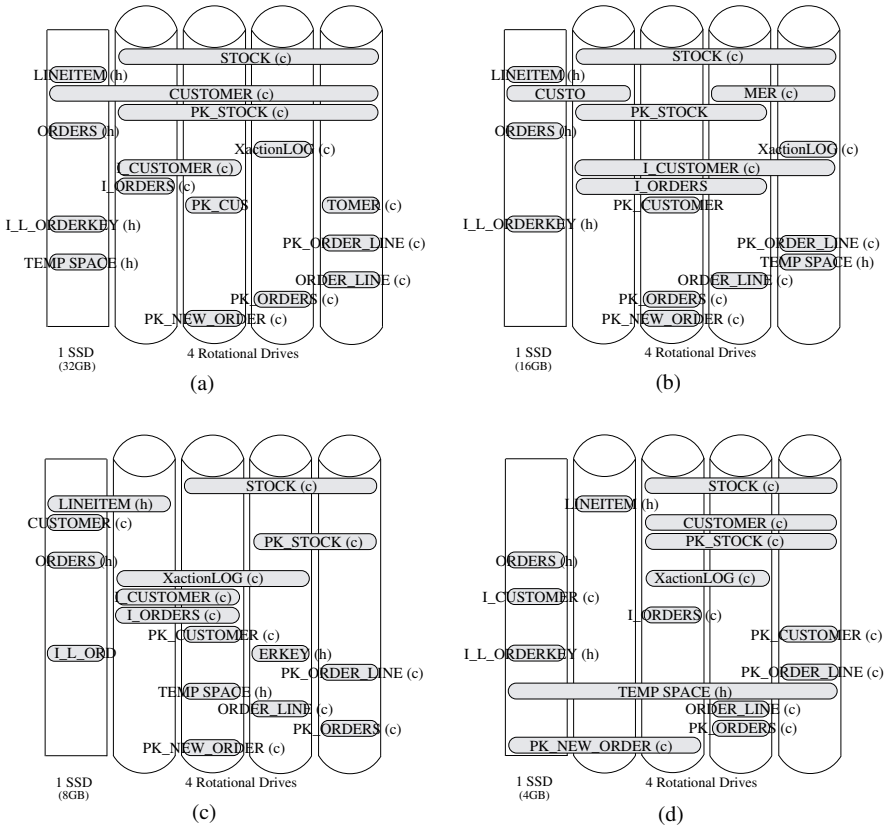


Figure 3.31: Optimized Layouts of the TPC-H (h) and TPC-C (c) Objects for the `Consolidation` Workload for the `4JBOD+SSD` Configuration. SSD capacity is set to (a) 32GB, (b) 16GB, (c) 8GB and (d) 4GB.

19 TPC-C objects is also around 10GB. Although the *volume* to the transaction log can grow up to 10-15GB in the experiments, only 16MB of the log is active[4] and the total size of the whole log does not exceed a couple of GBs at any time. Therefore, the 32GB SSD can host all of the TPC-H and TPC-C objects. We again consider scenarios in which the capacity of the SSD is reduced. Three other configurations are generated by setting the capacity of the SSD to 16GB, 8GB and 4GB. Figure 3.31 illustrates the advisor recommended layouts of the 15 most heavily-requested objects for the four scenarios.

Figure 3.32 summarizes the results. The most obvious observation is that both baseline layouts led to poor performance and the advisor recommended layouts resulted in much better performance for both OLAP and OLTP workloads. The reason behind the high performance degradation for baseline layouts is the SSD's low write I/O performance. Note that 65% of the requests are small-sized write-requests in the `OLTP8-90` workload. The layout advisor was aware of the distinct read/write characteristics of the devices and had knowledge about the I/O workload

[4]PostgreSQL transaction log is composed of regular files, each of which is at most 16MB.

Capacity of the SSD	Workload	SEE Baseline Performance	Baseline (all (objects on SSD) Performance	Optimized Layout Performance	Improvement Optimized vs. SEE
32GB	OLAP1-21	69387 sec 120.8 tpmC	60588 sec	4131 sec	16.8x
	OLTP8-90		62.9 tpmC	475.2 tpmC	3.9x
16GB	OLAP1-21		n/a	3880 sec	17.9x
	OLTP8-90			455 tpmC	3.8x
8GB	OLAP1-21			13452 sec	5.2x
	OLTP8-90			356 tpmC	2.95x
4GB	OLAP1-21	n/a	n/a	11739	5.9x
	OLTP8-90			431 tpmC	3.6x

Figure 3.32: Consolidation Scenario Performance Under the Baseline and Optimized Layouts for the 4JBOD+SSD Target Configuration.

of the objects, so it could do a better mapping of the objects. Considering the optimized layouts presented in Figure 3.32, we can conclude that the layout advisor tends to lay out TPC-H objects on the SSD and to distribute TPC-C objects across 4 rotational drives. This observation gave us a hint and we considered a third baseline layout in which TPC-H objects are laid out on the SSD and TPC-C objects are striped across 4 rotational drives. Note that this layout can be realized only when the SSD capacity is kept at 32GB. This new baseline layout has resulted in much better performance than the one observed when the initial two baseline layouts were used: the OLAP workload finished in 3994 seconds and the OLTP workload could run 387 tpmC. When compared to the optimized layout, the new baseline layout led to almost the same performance for the OLAP workload: 3994 seconds versus 4131 seconds. However, the optimized layout resulted in a better performance for the OLTP workload: 1.23x speedup over the new baseline layout, 475.2 tpmC versus 387 tpmC. This points to the difficulty of deciding what targets to use for which database system in a consolidated environment so that all database systems achieve good performance.

Lastly, another important observation that can be made from Figure 3.32 is the performance results for the optimized layouts obtained at SSD capacities of 8GB and 4GB. Note that the advisor picked a better layout for the 4GB case. Although the advisor could have picked the same layout for the 8GB case, it recommended a different layout. This shows that the optimizer does not guarantee the global optimum. For the 8GB case, the optimizer reached to a different solution because it started with a different initial layout. Nevertheless, the optimizer can be tuned or other initial layouts can be tried to reach to the same solution as the one obtained for the 4GB case.

Workload	N	M	Solver Time	Regularization Time	TOTAL Time
`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
2x`Consolidation`	80	10	47	12	59
3x`Consolidation`	120	10	340	40	380
4x`Consolidation`	160	10	590	72	662

Figure 3.33: Execution Time of the Layout Advisor (in seconds). N is the number of database objects, and M is the number of storage targets.

3.4.5 Optimization Time

Figure 3.33 shows the time required to recommend regularized layouts, for several different workloads. For each layout the total time required is reported, 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 costs for the `OLAP8-63` and `Consolidation` workloads. To recap, `OLAP8-63` accesses 20 TPC-H objects, and `Consolidation` workload both 20 TPC-H objects and 20 TPC-C 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, which are labeled 2x`Consolidation`, 3x`Consolidation`, and 4x`Consolidation` in Figure 3.33. 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 3.33 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.

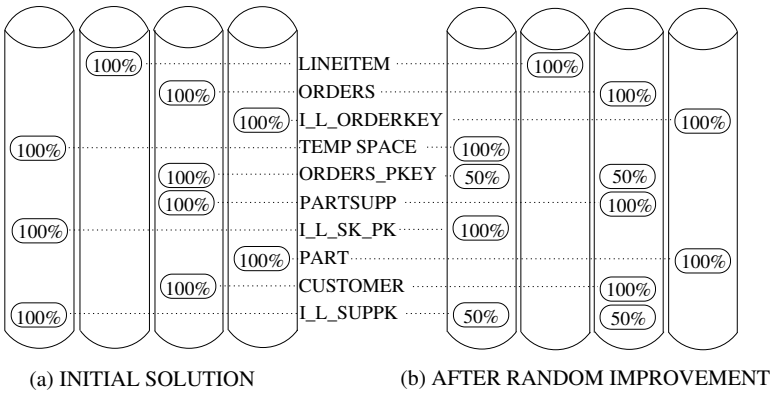


Figure 3.34: AutoAdmin Layout of TPC-H Database Objects for the `OLAP1-63` Workload

3.4.6 AutoAdmin Comparison

As part of the Microsoft AutoAdmin project, Agrawal et al [2] addressed a database layout problem that is similar to the one that we have considered, and they developed a tool for recommending layouts. Although that tool and the 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, the layout advisor expects statistical I/O workload parameters for each object. The advisor's 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 I/O 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 accessed 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, the layout advisor employs more expressive models. Unlike the AutoAdmin tool, I/O parallelism for individual database objects is not an explicit objective of the layout advisor, although the advisor often does distribute objects across multiple targets in order to balance load.

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 3.34 illustrates the layout generated by the AutoAdmin technique for the OLAP1-63 workload. Figure 3.34(a) shows the layout that results from the first step, in which each object is placed on a single target. Figure 3.34(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 the 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 the advisor 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 difference 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 Query 18 (see Figure 2.13). 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 3.34(b) is less balanced than that of Figure 3.23(a)(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 the advisor recommended layout did. The OLAP1-63 workload runs in 32634 seconds under AutoAdmin layout, as compared to 31789 seconds under the layout of Figure 3.23(a)(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 the 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 3.4.2, 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 3.34(b), compared to 16201 seconds for the SEE baseline and 13527 seconds (1.19x speedup) for the layout recommended by the advisor.

For the workloads we tested, the AutoAdmin implementation produced a layout more quickly than the 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. It has also been observed that graph partitioning (separat-

ing highly co-accessed objects) is the most time consuming step of the AutoAdmin layout algorithm.

3.5 Conclusion and Future Directions

In this chapter, we have presented a technique for recommending optimized layouts of database objects onto storage targets, such as rotational disk drives, SSDs, or RAID groups. However, the presented approach is general in that it is not limited to layout for database systems only. 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. More specifically, our objective is to minimize the maximum target utilization, and our black-box storage target models report target utilization for a set of workload descriptions. Although we have not experimented with other objectives, these black-box models could change to report other metrics such as request latency. However, we assume that the contributions of distinct workload descriptions into this metric are additive.

The presented 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 the proposed layout technique can provide layout recommendations to an administrator. This allows the administrators 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 [39] employ a feature called flexible volumes, or FlexVols [17]. 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. A future direction would be to explore how the layout techniques described in this thesis could be used to guide the storage system's dynamic allocation decisions as FlexVols grow. The NLP formulation of the layout problem allows us to experiment with the use of different allocation policies for individual flexible volumes and to isolate performance via allocation decisions rather than through other means such as bandwidth and resource reservations.

Another area for exploration is techniques for automating the construction of performance models for storage targets. Tabular models are well-suited for this purpose, as they are based on observations of the targets under load, rather than expert knowledge of the storage target internals. Nonetheless, it is a challenge to decide how to parametrize such models and how to design the calibration process.

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 [5] and DAD [8], which use heuristics to attack an even broader problem.

Chapter 4

Related Work

In the thesis, we have addressed two complementary and independent problems. First, we have introduced a method for estimating the storage workload of a database system. Next, we have described a technique which leverages such workload information to recommend a storage layout of database objects. In this chapter we survey other work related to these two problems.

4.1 Storage Workload Characterization

There are different ways to characterize storage workloads. One way to characterize storage workloads is to use a trace of I/O events. Traces are highly detailed and expressive. However, they are generally expensive to obtain, and it is hard to work with them. For example, to obtain the I/O workload of a DBMS, the DBMS needs to be populated and a realistic workload needs to be applied. The collected I/O trace may represent gigabytes of I/O requests and it will be specific to the current storage configuration. Therefore, it is also hard to generalize such traces.

Researchers have developed a more abstract way to characterize I/O workloads by analyzing such I/O traces. For example, as was discussed in Section 2.1.2, the Rome model [61] describes an I/O workload as a set of statistical parameters. It is a general purpose model intended to model storage workloads generated by any kind of storage client and for any configuration. The Rome model is utilized by a collection of storage management tools. As a result, the statistical parameters included in the model are determined by the storage system model found in those tools. For example, the Rome model needs to describe read/write mix, request rate, request size and sequentiality for each store defined in an I/O trace, and the pairwise temporal correlation between these stores. This raises another question: how much expressiveness do we need, or can we abstract the storage workload even further without losing accuracy?

Agrawal, Chaudhuri, Das, and Narasayya employed a similar model when addressing the problem of automating the layout of relational databases on a given

set of storage devices [2]. Some of the details of this work have been given in Section 3.4.6 (Microsoft AutoAdmin Comparison). Here, we will discuss in more detail the I/O model adopted in that work. The I/O model analyzes the query execution plans and abstracts the I/O workload as a set of statistical parameters similar to the Rome model. Internally, an access graph is used to characterize the storage workload resulting from a given database workload. Each database object is represented as a node in the graph and the weight of a node describes the estimated number of I/O requests (as fixed size pages) to the database object. An edge weight between two nodes characterizes the extent of pairwise co-access as the number of the pages that would be accessed simultaneously (e.g., as a result of the join of two objects). This is similar to the pairwise overlap fractions in our Rome-based descriptions. The methodology used to generate an access graph resembles the initial phase of our I/O estimation technique (query I/O request sequence estimation phase) described in Section 2.2.1. The execution plan of each SQL statement is obtained from the database query optimizer, and then each plan is analyzed in isolation to update the access graph. Agrawal et al. considered the layout problem as a database administration problem and targeted a simpler storage system than the ones modeled in storage configuration tools. The storage system is assumed to be non-consolidated and composed of a set of individual rotational hard drives. Moreover, some of the performance characteristics of storage devices are not modeled. For example, the complex relation between contention among concurrently accessed objects and I/O performance is not modeled. The I/O model is less expressive than the Rome model. For example, it makes no distinction between sequential and random I/O to an object and no distinction between reads and writes. In addition, this model ignores concurrency, and analyzes each query execution plan in isolation. Lastly, this model does not take into account the caching and prefetching effect on the storage workload, thus it may not accurately model the I/O workload as seen by the storage system.

Like the above work, our I/O estimation technique starts with a set of SQL statements and analyzes the query execution plans. However, our methodology is aware of the separation of the management of database and storage systems, and aims at obtaining the I/O workload that would be seen by the storage system. In contrast to the above work, our goal is to generate accurate database workload characterization to enable storage administrators to make informed decisions about layout and other related problems. Moreover, our technique produces I/O workload using a generic I/O model which is already utilized by storage configuration tools [7, 8, 27, 56]. All of these storage layer tools require storage workload characterizations, and can directly take advantage of our storage workload estimator.

4.2 Other Load Characterizations

A DBMS imposes loads on computing resources, such as CPU, memory, network and storage. Being able to characterize these loads enables system administrators to

make informed design decisions and maintain the system effectively. In this section, we will present some related work that involves resource load characterization.

Query optimizers have their own CPU and storage cost models. In general, CPU cost of an operator is directly proportional to the estimated number of tuples to be processed by the operator. Similarly, storage cost models are potentially simpler than the ones used in storage configuration tools. A query optimizer boils down storage workload into a single cost value for a given query. It is based on costing each page to be accessed by operators in an execution plan. The cost of accessing a page is determined by whether it is accessed sequentially or randomly using an unclustered index. Although this level of abstraction seems reasonable at the application level, it is hard to apply such an I/O model in a consolidated storage environment. This model ignores the correlation between concurrently running queries. For example, such a model is not expressive enough to capture the interference with a sequential stream if queries are running concurrently.

Now, we will present a few other systems that need to characterize resource loads in general. Narayanan, Thereska and Ailamaki describe a database resource advisor for predicting transaction response times and throughput based on end-to-end tracing [37]. Their technique relies on instrumentation and monitoring of *live* database systems. Like the technique described here, their approach seeks to identify a configuration-independent workload description with which to make model-based performance predictions. The model describes DBMS load as a sequence of events for each computing resource. For example, buffer pool activity is described as a sequence of page access requests. Similarly, storage activity is described as a block I/O trace. The history of resource activities allows the advisor to speculate about the impact of hypothetical changes in the underlying resources. However, because this approach relies on tracing a running database system, it has no means of speculating about the effects on the resource workloads of hypothetical changes in the database system workload or physical design. Our approach does accommodate such analyses.

Wasserman, Martin, Skillcorn and Rizvi [57] describe a resource-oriented workload characterization approach for database systems. They conduct characterization according to several resource-related attributes, such as CPU utilization and sequential and random I/O rates, as well as other properties such as join degree. Resource requirements of new workloads are computed by extrapolating from the previously monitored performance of queries. Our workload characterizations are more detailed, and they do not contain DBMS-specific attributes, such as join degree, that are not meaningful to the storage tier. More importantly, our estimation method does not depend on any previously collected data.

4.3 Database Design Tools

In the database tier, a variety of tools are available to address various aspects of the database physical design problem, such as choosing indexes and materialized

views [3, 62] and partitioning relations [3, 44]. These tools typically expect as input a database workload description similar to the one that is expected by our estimation technique. These tools are *complementary* to the workload estimation technique described in the thesis. As was discussed in Chapter 1, our storage workload estimation tool enables end-to-end solutions to database physical design and storage configuration problems. Figure 1.1 illustrates how existing database physical design tools and storage configuration tools could be combined using our I/O estimator to determine both a database physical design and an appropriate storage configuration for a given database workload.

4.4 Storage Layout

In this section, we will discuss work related to the data layout problem. Layout problem is addressed in several contexts. Some work focuses on laying out generic data (e.g., multimedia, web files), these are called file assignment problems (FAP) [14, 32]. Some approach the problem as a database administration problem, and specifically handle the layout of database objects [2, 45]. Data placement and partitioning in distributed systems has also a close resemblance to data placement in storage systems [46]. Lastly, there is research which considers the layout problem directly as a part of storage management and configuration task [5, 8, 9]. Although the layout problem is addressed in different contexts, there are essential similarities among the related work. Each work uses a cost or performance metric to optimize, has some form of a workload model which characterizes I/O behavior of the objects to be laid out, employs some performance model of the (storage) system, and lastly applies a search algorithm to traverse the solution space.

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 [14, 32]. 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. For solving FAP, heuristic algorithms are preferred, and bin packing algorithms are used to traverse solution space.

Rotem et al. [45] 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. [46] considered 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 data traffic is described as a graph by analyzing an SQL workload, and the edge weights in the graph describes the amount of data that needs to be carried from one node to another when executing queries. Thus, the problem becomes clustering the graph such that the sum of edges across clusters is minimized. 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] has already been described and analyzed in Section 3.4.6, and its I/O workload model has been discussed in this chapter. 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. The advisor's 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. As was explained, the AutoAdmin tool builds a graph representation of the I/O 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. This is done by partitioning the access graph such that the sum of edge weights between partitions is maximized. The second step further distributes objects across targets to increase I/O parallelism. This is done by using a greedy random improvement heuristic. The emphasis in the AutoAdmin work is on reducing interference among concurrently-accessed 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, the layout advisor employs more expressive models of I/O workload and storage system.

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 [5, 9, 48], culminating in the Disk Array Designer (DAD) [8]. A recent paper [61] 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.

Most of the related work in data layout rely on heuristic algorithms, such as bin-packing, greedy heuristics (e.g., random improvement) and genetic algorithms (e.g., simulated annealing), to traverse the solution space. The resulting layouts obtained through such heuristics are regular in our terminology. Moreover, these work restrict each data object to be laid out on a single target, which further reduces the size of the solution space. We formulate our problem as a non-linear programming problem and take advantage of state-of-art generic NLP solvers. Our layout advisor can generate both general and regular layouts.

Lastly, it is worth noting that the current practice relies mostly on generic rules of thumb, as discussed in Section 3.1.1. Database vendors give generic advice to database administrators on how to design the storage backend for databases. Vendor advice is usually generic because database vendors assume minimal knowledge about the database I/O workload. As a result, they see simplicity as an important layout property. All major vendors suggest distributing every database object across all available targets [41, 47]; an approach that is known as stripe-everything-everywhere (SEE). However, it is not clear how to apply SEE effectively in heterogeneous systems. It is not uncommon for a storage system to have storage targets varying in capacity and performance. Especially with the advent of SSDs, system heterogeneity has become a more commonly encountered situation. Database vendors offer heuristic layout hints in such situations, too. For example, they suggest what kinds of database objects to map to SSDs in order to get the most benefit out of them. The main idea of those heuristics is to isolate randomly accessed objects, such as unclustered indexes, onto SSDs [21].

Chapter 5

Conclusion

The thesis proposes a methodology which can be employed to generate optimized layouts for a given database physical design and database workload. It approaches the problem by dividing it into two complementary and independent pieces: storage workload estimation presented in Chapter 2 and workload-aware storage layout design presented in Chapter 3. More detailed conclusions and discussions of potential future research directions can be found at the ends of those two chapters.

In the first part of the thesis, we have focused on estimating the storage workload of a database system. The proposed estimation technique produces an I/O workload characterization that accurately models the storage workload, given a database workload and a database physical design. Storage workload models are generated in a form that is easily used by storage administration tools, such as configuration advisors. This makes it possible to combine database physical design tools and storage configuration tools and enable effective end-to-end design and configuration spanning both the database and storage system tiers. The feasibility of the proposed approach has been demonstrated by implementing it in the PostgreSQL DBMS. The experimental results suggest that the I/O workload estimator produces workload descriptions that are sufficiently accurate to be useful for predicting the performance of alternative storage configurations.

Storage workload information acts like glue between the database and storage tiers. It enables the storage tier to tune the storage layout with respect to the workload. In the second part of the thesis, we have focused on leveraging a storage workload characterization to recommend workload-aware storage layout. The proposed technique leverages input workload descriptions and storage device models to avoid potential interference among co-located objects, and to ensure that the recommended layout is balanced. It also uses the storage device models to ensure that the recommended layout reflects the distinct performance characteristics of each device. The layout optimization technique is incorporated into a database layout advisor which uses a generic non-linear programming (NLP) solver along with several heuristics that are specific to the layout problem. Formulating the problem as a generic NLP problem allows us to take advantage of state-of-the-art

NLP solvers rather than relying on guess work or human expertise. We demonstrate empirically that such a layout advisor can quickly recommend effective layouts over both homogeneous and heterogeneous storage target configurations.

In conclusion, this thesis makes several contributions. We formulate the storage workload estimation problem for relational database management systems. In our formulation, storage workloads are described in a domain-independent and configuration-independent language called Rome [60]. We present a technique for producing storage workload estimates and show its feasibility by implementing it in the context of the PostgreSQL DBMS. 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.

APPENDICES

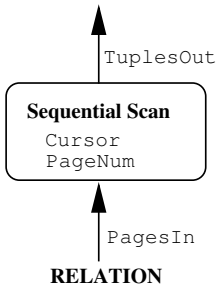
Appendix A

Data-Free Simulation of PostgreSQL Operators

Here, we give the illustrations of the simulation of all of the PostgreSQL query execution operators. More information on data-free operator simulation can be found in Section 2.2.1.

In these illustrations, the "ReadPage()" and "WritePage()" functions cause an I/O request to be *logged* in an I/O request sequence. Both functions accept two arguments representing an I/O request: a unique object identifier and an offset within the object. Note that request size is implicit because PostgreSQL uses an 8KB page size. There are only seven operators that can issue an I/O request. Three of them are scan operators that can be found only at the leaf level of an execution plan: table scan, index scan and tid scan operators. The other four operators can only be found in the inner nodes of a plan, and write into and read from temporary files: sort, hash, hash join and materialize. In the illustrations, curved boxes represent operators and they are annotated with the name of the operator and the names of state variables maintained by the simulation. Operator inputs and outputs are annotated with the names of PostgreSQL optimizer statistics and configuration parameters that are used by the simulator.

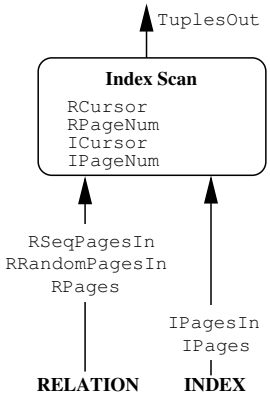
1. Table Scan



Init()	getNext()

Cursor := 0;
PageNum := 0;
	position := ceil(Cursor);
	Cursor += PagesIn/TuplesOut;
	for i := 1 to (ceil(Cursor)-position)
	  ReadPage(RELATION,PageNum);
	  PageNum += 1;

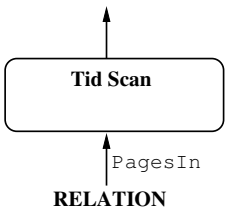
2. Index Scan



Init()	getNext()

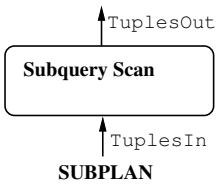
RCursor := 0;
RPageNum :=
  random(0,RPages-RSeqPagesIn);
ICursor :=
  random(0,IPages-IPagesIn);
IPageNum := ICursor;
	Iposition := ceil(ICursor);
	ICursor += IPagesIn/TuplesOut;
	Rposition := ceil(RCursor);
	RCursor += (RSeqPagesIn+RRandomPagesIn)/TuplesOut;
	for i := 1 to (ceil(ICursor)-Iposition)
	  ReadPage(INDEX,IPageNum);
	  IPageNum += 1;
	for i := 1 to (ceil(RCursor) - Rposition)
	  if Rposition < RSeqPagesIn
	  ReadPage(RELATION,RPageNum);
	  RPageNum += 1;
	  Rposition += 1;
	  else
	  pagenum := random in [0,..,RPages];
	  ReadPage(RELATION,pagenum);

3. Tid Scan



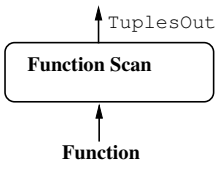
Init()	getNext()
/* Nothing */	PageNum := random(0, PagesIn); ReadPage(RELATION,PageNum);

4. Subquery Scan



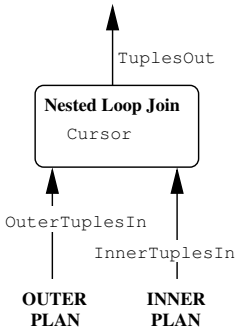
Init()	getNext()
Assert(TuplesIn == TuplesOut); Init(SUBPLAN);	getNext(SUBPLAN);

5. Function Scan



Init()	getNext()
Assert(TuplesOut == 1);	/* Nothing */

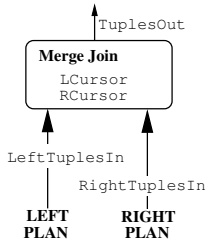
6. Nested Loop Join



Init()	getNext()

Cursor := 0;
Init(OUTERPLAN);
Init(INNERPLAN);
	position := ceil(Cursor);
	Cursor += OuterTuplesIn/TuplesOut;
	for i := 1 to (ceil(Cursor)-position)
	  getNext(OUTERPLAN);
	  Init(INNERPLAN);
	  for j := 1 to InnerTuplesIn
	  getNext(INNERPLAN);

7. Merge Join

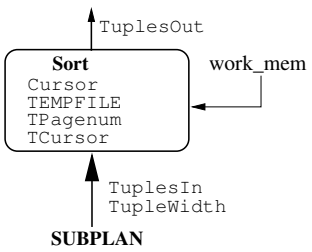


Init()	getNext()

LCursor := 0;
RCursor := 0;
Init(LEFTPLAN);
Init(RIGHTPLAN);
	Lposition := ceil(LCursor);
	Rposition := ceil(RCursor);
	LCursor += LeftTuplesIn/TuplesOut;
	RCursor += RightTuplesIn/TuplesOut;
	if(ceil(LCursor)-Lposition > ceil(RCursor)-Rposition)
	  for i := 1 to (ceil(RCursor)-Rposition)
	  getNext(RIGHTPLAN);
	  Rposition++;
	  for j := 1 to floor(LeftTuplesIn/RightTuplesIn)
	  getNext(LEFTPLAN);
	  Lposition++;
	else
	  for i := 1 to (ceil(LCursor)-Lposition)
	  getNext(LEFTPLAN);
	  Lposition++;
	  for j := 1 to floor(RightTuplesIn/LeftTuplesIn)
	  getNext(RIGHTPLAN);
	  Rposition++;
	/* read any remaining tuples */
	while(ceil(LCursor)-Lposition > 0)
	  getNext(LEFTPLAN);
	  Lposition++;
	while(ceil(RCursor)-Rposition > 0)
	  getNext(RIGHTPLAN);
	  Rposition++;

8. Sort

If the incoming tuples do not fit into the "work_mem" of the `Sort` operator, the external 6-way merge-sort algorithm is employed [29]. `Sort` operator uses a single temporary file to hold all the sorted runs. It is assumed that the size of the initial runs is $2 \cdot work\_mem$.

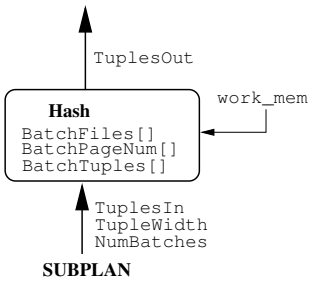


Init()	getNext()

Cursor := 0;
TEMPFILE :=
new TempFile();
TPageNum := 0;
TCursor := 0;
Init(SUBPLAN);
	if(TuplesIn · TuplesWidth < work_mem)
	/*Internal Sort*/
	while(Cursor < TuplesIn)
	getNext(SUBPLAN);
	Cursor++;
	else
	/*External Sort: 6-way merging*/
	if(Cursor == 0)
	/*Merge-Sort till the final sorted run is obtained*/
	/*Initial runs are written sequentially into the temp file*/
	while(Cursor < TuplesIn)
	getNext(SUBPLAN);
	Cursor++;
	if(Cursor%$\frac{8KB}{TupleWidth}$ == 0)
	WritePage(TEMPFILE, TPageNum);
	TPageNum++;
	/*Merge runs 6 by 6 into a new larger sorted run*/
	/*until the final sorted run is obtained*/
	num_runs := (TPageNum · 8KB)/(2 · work_mem);
	run_size := (2 · work_mem)/8KB;
	while(num_runs > 1)
	for(i := 1; 1 ≤ num_runs; i += 6)
	for j := 1 to 6
	runs[j].pagenum := random(0, TPagenum-run_size);
	runs[j].cursor := 0;
	new_run.pagenum := random(0, TPagenum-run_size*6);
	new_run.cursor := 0;
	/* read pages from each run sequentially */
	/* dump every 6 pages into new_run sequentially */
	num_runs := ceil(num_runs/6);
	run_size *= 6;
	/* At this point TEMPFILE has the final sorted run */
	position := ceil(TCursor);
	TCursor += TPagenum/TuplesOut;
	for i := position to ceil(TCursor)
	ReadPage(TEMPFILE, position);

9. Hash

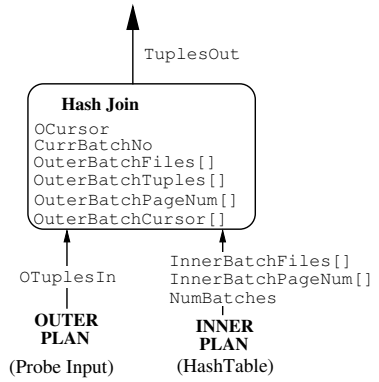
If the build-tuples will not fit in the "work_memory" then hash buckets are stored in temporary files (batches) such that each batch can fit into the operator's memory.



Init()	getNext()

if(NumBatches > 0)
  for i := 1 to NumBatches
  BatchFiles[i] := new TempFile();
  BatchTuples[i] := 0;
  BatchPageNum[i] := 0;
Init(SUBPLAN);
	for i := 1 to TuplesIn
	  getNext(SUBPLAN);
	  if(NumBatches > 0)
	  batch_no := random(1, NumBatches);
	  BatchTuples[batch_no]++;
	  if(BatchTuples[batch_no] · TupleWidth *geq* 8KB)
	  WritePage(BatchFiles[batch_no], BatchPageNum[batch_no]);
	  BatchPageNum[batch_no]++;
	  BatchTuples[batch_no] := 0; for i:= 1 to NumBatches
	  if(BatchTuples[i] > 0)
	  WritePage(BatchFiles[i], BatchPageNum[i]);
	  BatchPageNum[i]++;

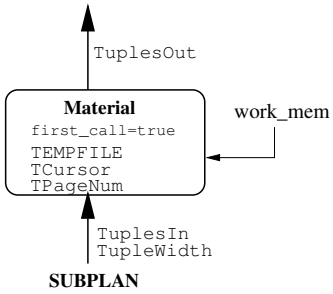
10. Hash Join



Init()	getNext()

Assert(type(INNERPLAN) == Hash);
Init(OUTERPLAN);
Init(INNERPLAN);
OCursor := 0;
CurrBatchNo := 1;
if(NumBatches > 0)
  for i := 1 to NumBatches
  OuterBatchFiles[i] :=
  new TempFile();
  OuterBatchTuples[i] := 0;
  OuterBatchPageNum[i] := 0;
/* Initiate Inner
  HashTable Generation */
getNext(INNERPLAN);
	if(OCursor < OTuplesIn)
	  position := ceil(OCursor);
	  OCursor += $\frac{OTuplesIn}{TuplesOut} \cdot \frac{1}{NumBatches+1}$;
	  for i := 1 to (ceil(OCursor)-position)
	  getNext(OUTERPLAN);
	  if(NumBatches > 0)
	  /*
	  Generate outer batches like Hash
	  operator does using:
	  OuterBatchFiles[], OuterBatchTuples[], and
	  OuterBatchPageNum[].
	  */
	else if(NumBatches > 0)
	  for i := 0 to InnerBatchPageNum[CurrBatchNo]
	  ReadPage(InnerBatchFiles[CurrBatchNo], i);
	  InnerBatchPageNum[CurrBatchNo] := 0;
	  position := ceil(OuterBatchCursor[CurrBatchNo]);
	  OuterBatchCursor[CurrBatchNo] += $\frac{OuterBatchPageNum[CurrBatchNo]}{TuplesOut/NumBatches}$;
	  while(position *leq* ceil(OuterBatchCursor[CurrBatchNo]))
	  ReadPage(OuterBatchFiles[CurrBatchNo], position);
	  position++;
	  if(OuterBatchCursor[CurrBatchNo] ==
	  OuterBatchPageNum[CurrBatchNo])
	  CurrBatchNo++;

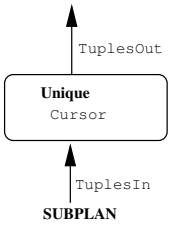
11. Materialize



Init()	getNext()

Assert(TuplesIn == TuplesOut);
if(first_call == true)
  Init(SUBPLAN);
  if(TuplesIn · TupleWidth > work_mem)
  TEMPFILE := new TempFile();
  TCursor := 0;
  TPageNum := 0;
	if(first_call == true)
	  first_call := false;
	  for i := 1 to TuplesIn
	  getNext(SubPlan);
	  if(TuplesIn · TupleWidth > work_mem)
	  Tposition := ceil(TCursor);
	  TCursor += $\frac{TuplesIn \times TupleWidth}{8KB} \cdot \frac{1}{TuplesOut}$;
	  for i := 1 to ceil(TCursor)-Tposition
	  WritePage(TEMPFILE, TPageNum);
	  TPageNum++;
	  TPageNum := 0;
	if(TuplesIn · TupleWidth > work_mem)
	  Tposition := ceil(TCursor);
	  TCursor += $\frac{TuplesIn \times TupleWidth}{8KB} \cdot \frac{1}{TuplesOut}$;
	  for i := 1 to ceil(TCursor)-Tposition
	  ReadPage(TEMPFILE, TPageNum);
	  TPageNum++;

12. Unique

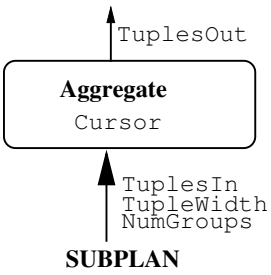


Init()	getNext()

Cursor := 0;
Init(SUBPLAN);
	position := ceil(Cursor);
	Cursor += TuplesIn ÷ TuplesOut;
	for i := 1 to ceil(Cursor)-position
	  getNext(SubPlan);

13. Aggregate

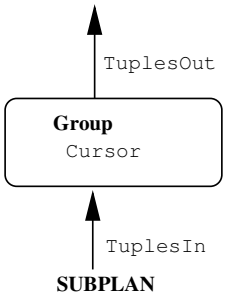
`Aggregate` operator is used for functions such as min, max, count. There are 3 strategies for aggregation: AGG_PLAIN, AGG_SORTED and AGG_HASHED. The later two are used for group aggregates. In AGG_SORTED strategy, the input is sorted, and in AGG_HASHED strategy the operator uses an internal hashtable.



Init()	getNext()

Assert((type(Aggregate)==PLAIN AND TuplesOut==1) OR
TuplesOut==NumGroups);
Init(SUBPLAN);
Cursor := 0;
	if(type(Aggregate) == PLAIN)
	for i := 1 to TuplesIn
	getNext(SubPlan);
	else
	position := ceil(Cursor);
	Cursor += TuplesIn/NumGroups;
	for i := 1 to ceil(Cursor)-position
	getNext(SUBPLAN);

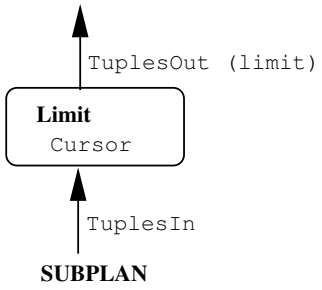
14. Group



Init()	getNext()

Cursor := 0;
Init(SUBPLAN);
	position := ceil(Cursor);
	Cursor += TuplesIn/TuplesOut;
	for i := 1 to ceil(Cursor)-position
	getNext(SubPlan);

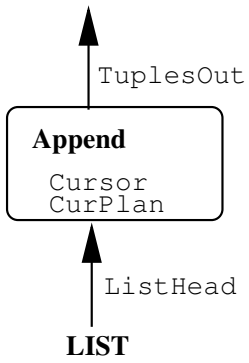
15. Limit



Init()	getNext()

Cursor := 0;
Init(SUBPLAN);
	if(Cursor < TuplesOut)
	getNext(SubPlan);
	Cursor++;

16. Append

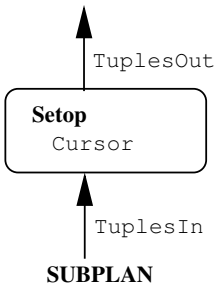


Init()	getNext()

CurPlan := ListHead;
Init(CurPlan);
Cursor := 0;
	if(Cursor ≥ CurPlan->TuplesIn)
	Cursor := 0;
	CurPlan := CurPlan->next;
	Init(CurPlan);
	getNext(CurPlan);
	Cursor++;

17. Setop

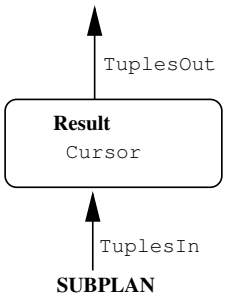
`Setop` operator is used for INTERSECT, INTERSECT ALL, EXCEPT, or EXCEPT ALL clauses.



Init()	getNext()

Assert(type(SUBPLAN)==SORT OR
type(SUBPLAN)==APPEND);
Init(SUBPLAN);
Cursor := 0;
	position := ceil(Cursor);
	Cursor += TuplesIn/TuplesOut;
	for i := 1 to ceil(Cursor)-position
	getNext(SUBPLAN);

18. Result



Init()	getNext()

Cursor := 0;
Init(SUBPLAN);
	position := ceil(Cursor);
	Cursor += TuplesIn/TuplesOut;
	for i := 1 to ceil(Cursor)-position
	getNext(SUBPLAN);

Appendix B

Measured and Estimated Rome Parameters for the WTPCH Workload

This appendix gives a direct comparison of all of the I/O workload parameters obtained from the actual workload trace and the estimated representative trace for the WTPCH workload at two concurrency levels, CL = 1 and CL = 5. In addition to the parameters that are explicitly specified in the Rome I/O model (see Figure 2.8), we also include some other parameters that are either used internally by the I/O workload analyzer Rubicon or derived from the existing Rome parameters.

Rubicon makes use of the total length of each projected trace to compute some of the Rome parameters (see Figure 2.8). The length of a trace is named "I/O Count" and it gives the number of I/O requests in the trace. The derived parameters are "page count", "percentage burst time" and "average request rate". Those parameters are derived as follows,

$$\begin{aligned} \text{Page Count} &= \frac{\text{I/O Count} \cdot \text{I/O Size}}{8KB} \\ \text{Percentage Burst Time} &= \frac{t_{on}}{t_{on}+t_{off}} \\ \text{Average Request Rate} &= \lambda \cdot \frac{t_{on}}{t_{on}+t_{off}} \end{aligned}$$

Page count is actually the I/O count normalized to 8KB pages. PostgreSQL normally issues 8KB I/O requests, but those requests can be merged into larger ones by the operating system. Since estimated and measured streams may have different I/O sizes, it may be deceiving to directly compare the I/O counts. That is why we include page count in the comparisons below. For example, at $CL = 1$, the measured I/O count of TempSpace object (i.e., TempSpace(write) and TempSpace(read)) is 528,631 I/O requests and the estimated I/O count is 487,754 I/O requests. However, we claimed that PostgreSQL has made a huge error by *overestimating* the number of tuples (thus pages) to be materialized in one of the execution plans (see Figure 2.13). In fact, considering the measured and the estimated page counts of this object, we see that the number of 8KB I/O requests to this object is highly

overestimated. While the measured page count is 1,428,047, the estimated page count is 2,827,729 pages. The other two derived parameters (percentage burst time and average request rate) are discussed in detail in Section 2.3.2.

Note that TempSpace is the only database object which receives write I/O requests. Therefore, we will not present two separate tables for the parameters that are distinguished by I/O type; these parameters are I/O size, I/O and page counts, and burst and average request rates. Instead, in the tables below, we will represent TempSpace using two rows, one for read requests (TempSpace(read)) and the other for write requests (TempSpace(write)). For the parameters that are indifferent to the I/O type, the TempSpace(read) row is used to represent the TempSpace object.

1. Results for CL=1

	I/O Size (KB)			I/O Count			Run Count		
object	meas.	est.	err.	meas.	est.	err.	meas.	est.	err.
lineitem	28.3	26.3	-6.9%	4607559	5048763	9.5%	696.4	788	13.1%
orders	24	26	8.1%	633895	651124	2.7%	1499.9	870.3	-41.9%
i_l_orderkey	9.8	9.2	-5.9%	496736	594813	19.7%	20.8	29.3	41.2%
orders_pkey	20.2	16.2	-19.7%	67380	102621	52.3%	178.5	85.7	-51.9%
partsupp	50.4	33	-34.3%	50611	80336	58.7%	2806.8	1172.4	-58.2%
i_l_suppkey_partkey	13.9	8	-42.6%	22110	80151	262.5%	1.6	1	-37.8%
customer	102.9	126.5	22.9%	21589	15971	-26%	1465.2	1150.3	-21.4%
part	104.3	127.5	22.3%	15598	12994	-16.6%	1793.7	1478.8	-17.5%
i_l_suppkey	32.8	19.9	-39.3%	8787	7891	-10.1%	3.4	1.9	-42.6%
i_ps_partkey	59	12.3	-79%	6583	30061	356.6%	195	21	-89.1%
i_o_custkey	97.2	16.7	-82.7%	5405	28038	418.7%	1276.9	76.5	-94%
supplier	29.4	22.8	-22.5%	4026	4981	23.7%	40.4	61.9	53.2%
TempSpace(write)	19.5	40	105.1%	498813	392568	-21.2%			
TempSpace(read)	56.9	72.6	27.6%	29818	95186	219.2%	30.1	32.2	6.8%
part_pkey	98.7	125.4	27%	1484	1167	-21.3%	219	194.5	-11.2%
i_l_partkey	10	8	-20.2%	1228	27926	2174.1%	1.1	1	-14.5%
customer_pkey	96.6	125.2	29.5%	758	585	-22.8%	133.3	146.2	9.7%
supplier_pkey	8.5	8.1	-4.6%	289	303	4.8%	1	1	-6.4%
i_o_orderdate	71.3	121.5	70.4%	173	100	-42.1%	40.6	48	18.2%
nation	8	8	0%	12	9	-25%	1	1	0%
region	8	8	0%	6	4	-33.3%	1	1	0%

I/O Size, I/O Count and Run Count Parameters

	Request Rate (req/s)			Avg. Burst(ON) Time (sec)			Avg. OFF Time (sec)		
object	meas.	est.	err.	meas.	est.	err.	meas.	est.	err.
lineitem	214.5	265.2	23.6%	894.6	346	-61.3%	88	82.5	-6.2%
orders	55.4	79	42.5%	120.2	196.1	63%	128.4	372	189.7%
i_l_orderkey	32.1	55.4	72.2%	965.4	511.2	-47%	503.1	607.4	20.7%
orders_pkey	8.8	17.4	96.6%	229.7	345.3	50.3%	482.2	1036.4	114.9%
partsupp	17.9	89.1	397.8%	403.7	112.6	-72.1%	2583.9	2509.9	-2.8%
i_l_suppkey_partkey	8.3	203.2	2334.6%	1324.1	197.1	-85.1%	6949.6	7698.8	10.7%
customer	77.3	43.5	-43.7%	0.7	1.1	41.7%	65.4	70	7.1%
part	71.2	17.6	-75.2%	0.8	6.5	721.2%	85.2	201.3	136.1%
i_l_suppkey	1.4	1.4	4.2%	14	445.3	3060.9%	39.2	1395.8	3458.1%
i_ps_partkey	8.9	35.2	295.2%	1.3	142	10502.9%	41.1	3234	7757.2%
i_o_custkey	14	70.4	401.2%	9.6	30.6	218.4%	563.7	1649.5	192.6%
supplier	19.3	11	-42.7%	1	0.9	-8.8%	113	47.7	-57.8%
TempSpace(write)	501.4	106.4	-78.7%						
TempSpace(read)	29.9	25.8	-13.8%	0.6	6.4	871.2%	14.8	34.4	131.7%
part_pkey	24	15.7	-34.5%	1.4	1.4	-4.7%	545	433.6	-20.4%
i_l_partkey	10.7	25.1	134.7%	114.6	1110.6	868.8%	11691.3	11190	-4.2%
customer_pkey	40.9	50.9	24.5%	0.2	0.1	-25%	244.5	308.9	26.3%
supplier_pkey	80.1	52.9	-33.9%	1.8	2.8	58.8%	7831.2	7828.3	-0%
i_o_orderdate	384.9	15.7	-95.9%	0	0.4	3900%	469.9	1381.4	193.9%
nation	188.7	582.4	208.6%	0	0	-100%	1807.4	2349	29.9%
region	304.5	582.4	91.2%	0	0	0%	3356.7	4698.1	39.9%

Request Rate and Burst(ON)/Inter-Burst Gap(OFF) Durations

	Page Count			Percentage ON Time (%[0,100])			Avg. Request Rate (req/s)		
object	meas.	est.	err.	meas.	est.	err.	meas.	est.	err.
lineitem	16324437.5	16651919.8	2%	91	80.7	-11.3%	195.3	214.1	9.6%
orders	1904934.9	2116709.3	11.1%	48.3	34.5	-28.6%	26.8	27.2	1.7%
i_l_orderkey	611036.2	688042.9	12.6%	65.7	45.6	-30.4%	21.1	25.3	19.7%
orders_pkey	170917.5	208987.5	22.2%	32.2	24.9	-22.5%	2.8	4.3	52.2%
partsupp	319092.7	332307.8	4.1%	13.5	4.2	-68.2%	2.4	3.8	58.2%
i_l_suppkey_partkey	38581.8	80160.7	107.7%	16	2.4	-84.4%	1.3	5	279.1%
customer	277784.4	252720.7	-9%	1.1	1.5	31.8%	0.9	0.6	-26%
part	203379.4	207215.5	1.8%	0.9	3.1	239.9%	0.6	0.5	-16.4%
i_l_suppkey	36055.4	19635.9	-45.5%	26.4	24.1	-8.4%	0.3	0.3	-2.7%
i_ps_partkey	48594.6	46493.2	-4.3%	3.1	4.2	33.4%	0.2	1.4	428.5%
i_o_custkey	65699.9	58827.7	-10.4%	1.6	1.8	8.6%	0.2	1.2	433.3%
supplier	14826.2	14205.4	-4.1%	0.8	1.9	113.8%	0.1	0.2	23.5%
TempSpace(write)	1215856.6	1962840	61.4%				21.2	16.7	-21.3%
TempSpace(read)	212191.1	864888.9	307.5%	4.2	15.6	268.9%	1.2	4	218.8%
part_pkey	18315.9	18305.8	-0%	0.2	0.3	19.6%	0	0	-16.6%
i_l_partkey	1538.8	27926	1714.6%	0.9	9	829.9%	0.1	2.2	2170%
customer_pkey	9156.9	9155.9	-0%	0.082	0.049	-40.6%	0.03	0.03	0%
supplier_pkey	307.9	307.9	0%	0.023	0.037	58.9%	0	0.02	0.02%
i_o_orderdate	1541.9	1518.9	-1.4%	0.002	0.029	1260.3%	0.01	0.008	-18%
nation	12	9	-25%	0	0	0%	0	0	0%
region	6	4	-33.3%	0	0	0%	0	0	0%

Derived Parameters: Page Count, Percentage Burst Time and Avg. Request Rate

2. Results for CL=5

	I/O Size (KB)			I/O Count			Run Count		
object	meas.	est.	err.	meas.	est.	err.	meas.	est.	err.
lineitem	28.7	22.4	-21.9%	6235449	7815489	25.3%	1.8	1.5	-16.6%
orders	19.2	22.6	17.7%	1308526	1183903	-9.5%	163.7	5.2	-96.8%
i_l_orderkey	10.5	8.5	-19%	1050374	1504873	43.2%	1	1	0%
orders_pkey	20.3	11.9	-41.3%	91843	238573	159.7%	31.4	9.6	-69.4%
partsupp	53.8	47.5	-11.7%	76974	151493	96.8%	2279.7	1292.8	-43.2%
i_l_suppkey_partkey	13.9	8	-42.4%	33136	136689	312.5%	1.6	1	-37.5%
part	100.4	126	25.4%	31610	25783	-18.4%	1469.7	1026.8	-30.1%
customer	89.5	125.5	40.2%	31257	21076	-32.5%	770.9	805.2	4.4%
i_l_suppkey	27.8	20	-28%	10041	11939	18.9%	2.1	1.5	-28.5%
i_ps_partkey	57.8	11	-80.9%	10001	56070	460.6%	140.9	11.7	-91.6%
i_o_custkey	71.1	18	-74.6%	9566	36656	283.1%	937.6	87.5	-90.6%
supplier	22.9	21.4	-6.5%	7407	8997	21.4%	33.5	22.7	-32.2%
TempSpace(write)	16.8	50.2	198.8%	679085	348835	-48.6%			
TempSpace(read)	52.3	62	18.5%	74648	166207	122.6%	17.2	37.2	116.2%
customer_pkey	22	26.3	19.5%	5376	4711	-12.3%	84.5	96.8	14.5%
part_pkey	100	118.8	18.8%	2196	1852	-15.6%	217.4	181.7	-16.4%
i_l_partkey	10	8	-20%	1228	31060	2429.3%	1.1	1	-9%
i_o_orderdate	14.5	9.3	-35.8%	1190	3520	195.7%	96.8	185.4	91.5%
supplier_pkey	8.5	8.1	-4.7%	432	453	4.8%	1	1	0%
nation	8	8	0%	18	19	5.5%	1	1	0%
region	8	8	0%	8	10	25%	1	1	0%

I/O Size, I/O Count and Run Count Parameters

	Request Rate (req/s)			Avg. Burst(ON) Time (sec)			Avg. OFF Time (sec)		
object	meas.	est.	err.	meas.	est.	err.	meas.	est.	err.
lineitem	349.4	516.9	47.9%	17843.7	408.5	-97.7%	1	15.5	1450%
orders	82.2	101.4	23.3%	131.4	12.3	-90.6%	16.1	4.2	-73.9%
i_l_orderkey	58.8	100.5	70.9%	17842.7	356.3	-98%	1	17.2	1620%
orders_pkey	7.5	37.9	405.3%	113.7	3.5	-96.9%	53	5.2	-90.1%
partsupp	10.4	39.2	276.9%	613.3	203	-66.9%	873.6	622.8	-28.7%
i_l_suppkey_partkey	4.3	133.8	3011.6%	761.6	170.2	-77.6%	1022.7	2444.9	139%
part	51.6	16.4	-68.2%	1.1	2.1	90.9%	31.7	19.7	-37.8%
customer	28.2	23.7	-15.9%	0.7	1.5	114.2%	10.6	25.3	138.6%
i_l_suppkey	1.8	1.1	-38.8%	2.6	184.5	6996.1%	5.9	98.9	1576.2%
i_ps_partkey	9.3	21.6	132.2%	0.9	17.9	1888.8%	14.6	91	523.2%
i_o_custkey	12.5	83.1	564.8%	0.6	1.1	83.3%	14.5	40	175.8%
supplier	23.2	15.4	-33.6%	0.3	0.4	33.3%	18.2	11.2	-38.4%
TempSpace(write)	206.3	42.7	-79.3%						
TempSpace(read)	22.6	20.3	-10.1%	1.4	17.6	1157.1%	6.2	16.2	161.2%
customer_pkey	2.1	4.3	104.7%	5.1	9.3	82.3%	31.5	123.7	292.6%
part_pkey	21.5	11.5	-46.5%	1.5	1.2	-20%	268.8	123.2	-54.1%
i_l_partkey	4.7	20.7	340.4%	260.2	500	92.1%	8792.5	3548	-59.6%
i_o_orderdate	73.5	6.1	-91.7%	0.1	0.4	300%	29.7	11.4	-61.6%
supplier_pkey	18.1	25.8	42.5%	4.7	5.8	23.4%	2970.2	3918.7	31.9%
nation	109	2028.4	1760.9%	0.01	0	0%	991.3	825.9	-16.6%
region	112.2	2028.4	1707.8%	0	0	0%	1982.8	1426.5	-28%

Request Rate and Burst(ON)/Inter-Burst Gap(OFF) Durations

	Page Count			Percentage ON Time (%)			Avg. Request Rate (req/s)		
object	meas.	est.	err.	meas.	est.	err.	meas.	est.	err.
lineitem	22380481.8	21886612.9	-2.2%	100	96.3	-3.6%	349.4	498	42.5%
orders	3152793.7	3356752.3	6.4%	89	74.3	-16.4%	73.2	75.4	2.9%
i_l_orderkey	1378872.3	1609765.8	16.7%	100	95.3	-4.6%	58.8	95.9	62.9%
orders_pkey	234114.4	357684.7	52.7%	68.2	40.1	-41.1%	5.1	15.2	195.3%
partsupp	518522.1	900081.4	73.5%	41.2	24.5	-40.3%	4.3	9.6	123.8%
i_l_suppkey_partkey	57874.7	136705.6	136.2%	42.6	6.5	-84.7%	1.8	8.7	368.2%
part	396795.7	406308.8	2.3%	3.4	9.9	190.7%	1.7	1.6	-7.3%
customer	350073	330848.4	-5.4%	6.2	5.6	-9.1%	1.7	1.3	-23.4%
i_l_suppkey	34958.4	29851.8	-14.6%	31.1	65	109%	0.5	0.7	37.5%
i_ps_partkey	72259.4	77561.6	7.3%	5.9	16.4	176.6%	0.5	3.5	537.5%
i_o_custkey	85090.9	82851.8	-2.6%	4.2	2.8	-33.9%	0.5	2.3	333.3%
supplier	21276.1	24162.9	13.5%	1.7	3.7	111.8%	0.4	0.5	35.7%
TempSpace(write)	1429377.7	2192823.1	53.4%				38	22.2	-41.5%
TempSpace(read)	488100.6	1289970.8	164.2%	18.4	51.9	181.1%	4.1	10.5	153.3%
customer_pkey	14844.3	15516.6	4.5%	13.9	7	-49.9%	0.3	0.3	0%
part_pkey	27468.7	27508.9	0.1%	0.57	1	79.2%	0.1	0.1	0%
i_l_partkey	1538.8	31060	1918.3%	2.8	12.3	329.7%	0.1	2.5	1728.5%
i_o_orderdate	2157.4	4127.5	91.3%	0.1	3.7	3574.6%	0.07	0.2	214.2%
supplier_pkey	461.9	461.9	0%	0.16	0.15	-6.9%	0.03	0.04	33.3%
nation	18	19	5.5%	0	0	0%	0	0	0%
region	8	10	25%	0	0	0%	0	0	0%

Derived Parameters: Page Count, Percentage Burst Time and Avg. Request Rate

Appendix C

Generating Performance Look-up Tables for Storage Devices

We have two cost models (i.e., look-up tables) for each storage device *type*: one model for read requests and the other for write requests. All devices of the same type can utilize the same cost model. As explained in Section 3.2.2.2.1, the load imposed by an I/O workload on a storage device depends on four parameters: its request rate, its request size, its run count (sequentiality), and a contention factor. The first three parameters are the direct properties of the workload, given as input. The contention factor represents the extent of the interference of the remaining workloads with the given workload. Hence, the load of workload W_i on a storage device of type T_j becomes,

$$\mu_i = \lambda_i^r \cdot \mathcal{COST}_{T_j}^{\text{read}}(B_i^r, Q_i, \chi_i) + \lambda_i^w \cdot \mathcal{COST}_{T_j}^{\text{write}}(B_i^w, Q_i, \chi_i)$$

Here, $\mathcal{COST}_{T_j}^{\text{read}}(B_i^r, Q_i, \chi_i)$ and $\mathcal{COST}_{T_j}^{\text{write}}(B_i^w, Q_i, \chi_i)$ are the look-up operations for read and write request costs for a given request size, run count, and contention factor. Below, we will explain how we construct a tabular model and how we implement look-up operation.

C.1 Look-Up Table Construction

A look-up table is populated by measuring I/O request times for various workloads. Workloads are obtained by using a synthetic workload generator which generates an I/O workload corresponding to given values for the three workload parameters: request size, run count and contention factor. In general, deciding on what values to use when populating a look up table is a difficult job because there could be too many parameters and each parameter can take a large set of values. Covering the whole parameter space would be infeasible. However, by applying the domain knowledge, we can shrink the set of possible values dramatically.

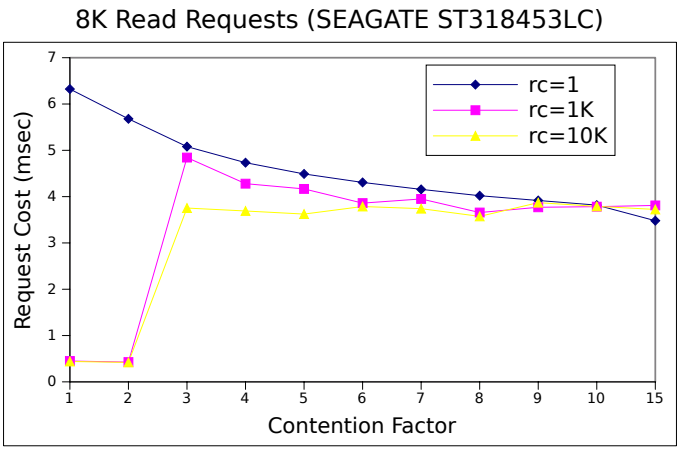


Figure C.1: Request Cost versus Contention Factor at Different Run Counts.

Our testbed uses PostgreSQL database system and Linux operating system. PostgreSQL issues 8KB requests and the Linux version installed in our system puts a hard limit of 512KB on the maximum size of any block I/O request. This I/O size limit depends on the capability of the underlying device controller, which negotiates with the kernel when setting this limit. Although PostgreSQL issues 8KB requests, because of prefetching and request merging that happens at the scheduler's queue, it is not unusual to observe larger request sizes. As a result, the workload generator sets request sizes to the powers of two within [8KB, 512KB].

As for the run count parameter, we limit the max run count value to 10K. To recap, run count is the average number of requests in a sequential run. Thus, a purely random workload has a run count value of 1, which means no two consecutive requests are sequential and 100% of requests are random. As the run count value increases, the number of random jumps in a workload decreases. For example, a run count value 10K means that there is a random seek after every 10K requests, thus 0.01% of the requests are random. As a result, there is not a big performance difference between I/O workloads whose run counts are 10K, 100K and even 1000K. Figure C.1 illustrates this fact; the costs of requests belonging to workloads whose run counts are 1K and 10K are almost the same. The same observation can also be made for the contention factor, as the contention factor increases request cost is flattened at a constant value, so there is no need to generate measurements for higher contention factors. Figure C.2 summarizes the values the workload generator uses for the three parameters.

Each of our three dimensional look-up tables include $7 \times 26 \times 11 = 2002$ measurements. Measurements are taken by generating I/O workloads for all combinations of the values given in Figure C.2 and measuring the I/O request times. For example, the I/O request time for the point {request size:8KB, run count:100, contention factor:9} is measured by running 9 identical processes (i.e., request streams), each of which issues a single request at a time. Each request has a size of 8KB, and each

Parameter	Values
Request Size	8, 16, 32, 64, 128, 256 and 512 KB.
Run Count	1 2 3 4 5 6 7 8 9 10 13 15 20 25 50 75 100 200 300 400 500 750 1000 2000 5000 10000
Contention Factor	1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and 15

Figure C.2: Values Used for the Look-up Table Parameters

	Read	Write
SEAGATE ST318453LC	46 hrs	37 hrs
Patriot PE32GS25SSDR	13 hrs	49 hrs

Figure C.3: LookUp Table Generation Times

stream maintains a run count of 100. In the end of the run, we obtain the number of requests completed and the total time in which the target device has been busy with serving I/O requests during the measurement period. Here, we depend on the statistics maintained by the Linux kernel to measure the busy duration of a device. The Linux kernel exposes several statistics for storage devices (including device use time) in the `/proc/diskstats` file. Finally, we can measure the cost of a single request as $\frac{\text{Total Time the Device was Busy}}{\text{Number of Requests Completed}}$. Thus, this measured cost represents the device service time for an I/O request which belongs to a workload with a request size of 8KB, run count of 100 and a contention factor of 9.

In our system, we have two types of storage devices to hold database objects: four SEAGATE ST318453LC rotational hard drives [51] and a single Patriot PE32GS25SSDR solid-state drive [33]. Thus, we need to generate four tables: two tables per device type. Each measurement takes at least 5 seconds. However, since the workload generator ensures that each request stream completes at least three sequential runs, some measurements for which the run count parameter is given a large value can take more time to finish. Figure C.3 presents the run time of the whole look-up table generation process for both devices.

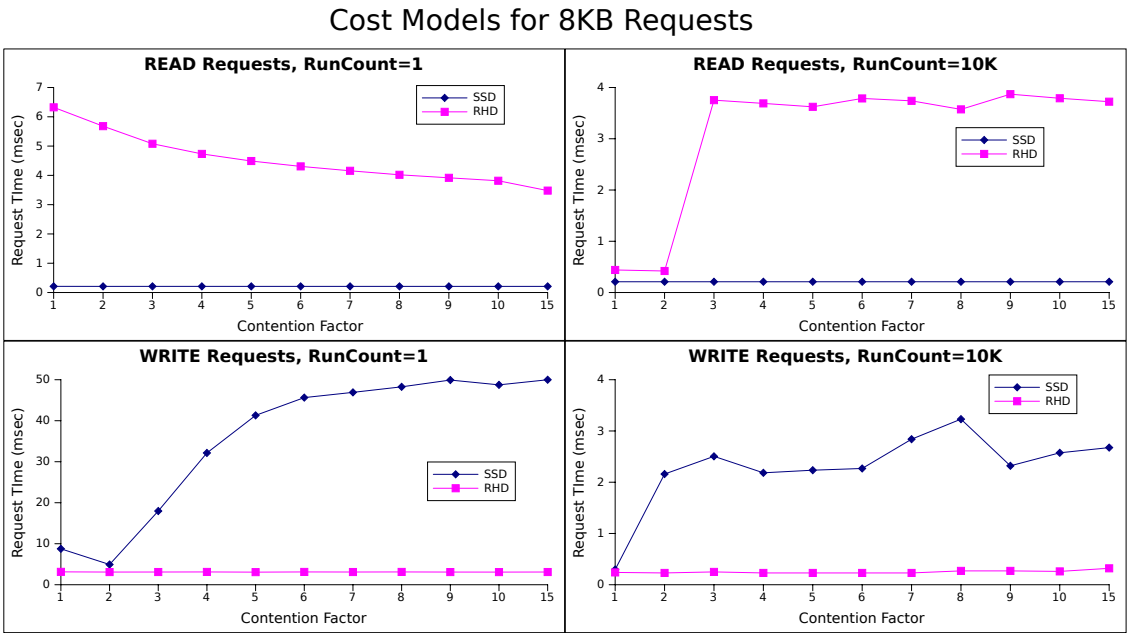
Lastly, in order to show that performance of storage devices can be highly non-linear with respect to workload parameters and that different device types can have highly different performance characteristics, we present slices from our look-up tables. Figure C.4 shows the request service times (in milli-seconds) of 8KB and 128KB requests for two device types with respect to contention factor under four different workloads: read-random, read-sequential, write-random, and write-sequential. For small-sized read requests (e.g., 8KB), solid-state derive (SSD) has better performance than rotational hard drive (RHD). The RHD has comparable read performance only when the workload is highly sequential and only when it is not interfered with by other workloads. On the other hand, the RHD has better performance for write requests, especially when the workload is random. As the re-

quest size increases (e.g., 128KB), we observe similar behavior for the read requests and random-write requests. However, now, the SSD's write performance becomes better than RHD when the workload is sequential.

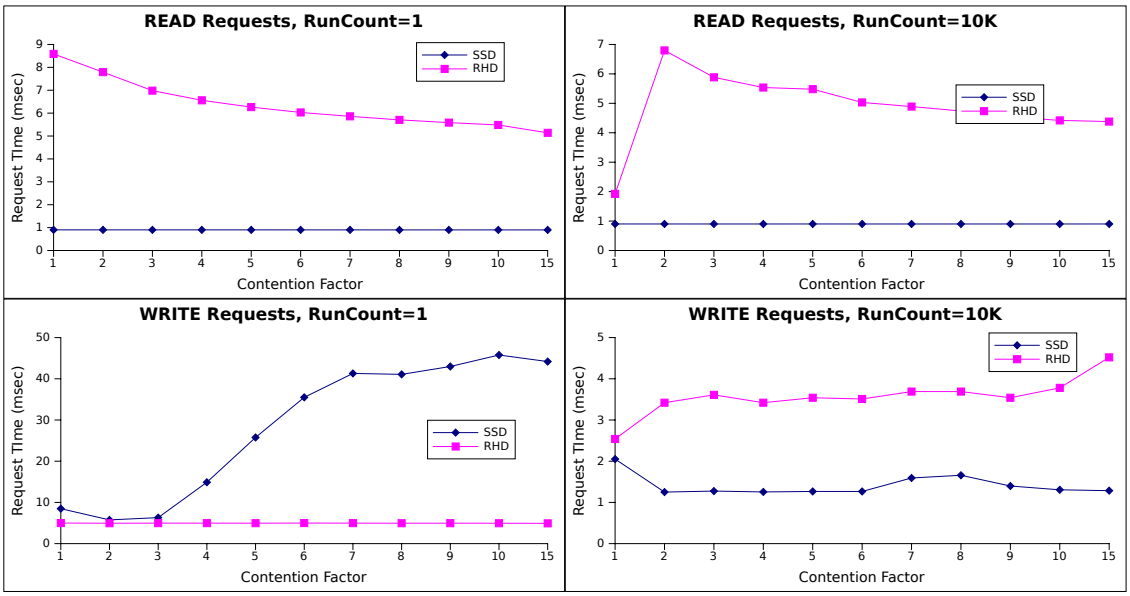
C.2 Using the Tabular Cost Model

For a given request size, run count and contention factor triplet, the look-up operation may not always locate an exact match within the cost table. For example, a workload may have the following parameters: (request size:40KB, run count:35, contention factor:3.3). In such cases, the look-up operation interpolates among the nearest entries in the table. It first locates the nearest entries for each of the three parameters. If there is no exact match in any of three dimensions, this look-up ends up with 8 nearest points. Assume the look-up operation for (rs, rc, cf) results in 8 nearest points: $(rs_i, rc_j, cf_k), 1 \leq i, j, k \leq 2$. The cost model interpolates one dimension at a time to approximate the performance point at (rs, rc, cf):

1. contention factor dimension: $(rs_i, rc_j, cf) = \frac{\sum_{k=1}^{2}(rs_i, rc_j, ql_k)}{2},\ 1 \leq i, j \leq 2$
2. run count dimension: $(rs_i, rc, cf) = \frac{\sum_{j=1}^{2}(rs_i, rc_j, ql)}{2},\ 1 \leq i \leq 2$
3. request size dimension: $(rs, rc, cf) = \frac{\sum_{i=1}^{2}(rs_i, rc, ql)}{2}$



(a) Request Size = 8KB



(b) Request Size = 128KB

Figure C.4: Cost Models for a Rotational Hard Drive and an SSD: The request service times of (a) 8KB and (b) 128KB I/O requests for two device types (RHD and SSD) with respect to contention factor under four workloads defined by I/O type (read and write) and run count (random:1 and sequential:10K).

Appendix D

Modeling the Layout Problem Using AMPL

Below is the AMPL code which implements the storage system model introduced in Section 3.2.2. AMPL allows us to represent the optimization problem in a standard way and thus to benefit from generic NLP solvers which accept problems stated in AMPL.

```
#
# This is the storage system model:
#

# These are the user-defined functions (written in C)
# which are employed to look-up the cost table. Definitions of
# these functions should be in the amplfunc.so shared library.
function tableLookUp_r;
function tableLookUp_w;

#
# PARAMETERS (inputs to the problem):
#
param N > 0, <= 500; # Number of Objects
param M > 0, <= 100; # Number of Targets

# For each target (target parameters):
param c{i in 1..M} > 0;           # Storage Capacity
param D{i in 1..M} > 0 integer;   # Number of Storage Devices
param U{i in 1..M} >= 8, <= 128 integer;# Stripe Unit (in Kbytes)
param T{i in 1..M} > 0 integer;   # Device Types

# For each object (object and workload parameters):
param s{i in 1..N} > 0;           # Size of Objects
param R{i in 1..N} >= 0, <= 512; # Avg. Read Request Size
param W{i in 1..N} >= 0, <= 512; # Avg. Write Request Size
param lambda_r{i in 1..N} >= 0;  # Avg. Read Request Rate
param lambda_w{i in 1..N} >= 0;  # Avg. Write Request Rate
```

```
29 param Q{i in 1..N} >= 1;          # Run Count
30 param O{i in 1..N, j in 1..N} >= 0, <= 1; # Pairwise Overlap Fraction Matrix
31
32 # Stripe Unit Used by the Layout Model:
33 param layoutSU >= 8, <= 256, default 128;
34
35 #
36 # LAYOUT and TARGET Model Transformations:
37 #    pre-computed parameters and variables
38 #
39
40 # Request Size Scale Factor: using R, U and D.
41 #   A large request within a target is broken into smaller requests:
42 param SF_r{i in 1..N, j in 1..M} := if R[i] > U[j] then 1/D[j] else 1;
43 param SF_w{i in 1..N, j in 1..M} := if W[i] > U[j] then 1/D[j] else 1;
44
45 # Request Size Transformation
46 param rRS{i in 1..N, j in 1..M} := SF_r[i,j]*R[i];
47 param wRS{i in 1..N, j in 1..M} := SF_w[i,j]*W[i];
48
49 # Request Rate Transformation
50 param rRR{i in 1..N, j in 1..M} := lambda_r[i]/(SF_r[i,j]*D[j]);
51 param wRR{i in 1..N, j in 1..M} := lambda_w[i]/(SF_w[i,j]*D[j]);
52
53 # Read Probability Used by Run Count Transformation
54 param read_prob{i in 1..N} := lambda_r[i]/(lambda_r[i] + lambda_w[i]);
55
56 # Total Request Rate: used by contention factor calculation
57 param total_rate{i in 1..N, j in 1..M} :=
58             lambda_r[i]/SF_r[i,j] + lambda_w[i]/SF_w[i,j];
59
60 # To be Used by Run Count Transformation:
61 #       a sequential run can not be larger than the layout stripe unit
62 #       if the stream is divided into more than one target.
63 param avg_rs{i in 1..N} := read_prob[i]*R[i] + (1-read_prob[i])*W[i];
64 param layout_model_rc_min_limit{i in 1..N} := min(Q[i], layoutSU/avg_rs[i]);
65
66 #
67 # VARIABLES
68 #
69
70 # DECISION VARIABLES:
71 var conf_matrix{i in 1..N, j in 1..M} >= 0, <= 1; # LAYOUT MATRIX (L)
72 var max_util >= 0
73
74 # Definition of the Contention Factor
75 var cf{i in 1..N, j in 1..M} =
76     sum{k in 1..N} ( O[i,k]  *  ((conf_matrix[k,j]/(conf_matrix[i,j])) *
77                                  (total_rate[k,j]/total_rate[i,j])) );
78
79 # Adjusted Run Count (layout model): R.C. of the i-th wkld seen by the j-th target.
80 var Q_ij{i in 1..N, j in 1..M} = max(layout_model_rc_min_limit[i],
81                                      Q[i] * conf_matrix[i,j]);
82
```

```
83  # Adjusted Run Count: (RAID transformation)
84  param target_avg_rs{i in 1..N, j in 1..M} :=
85              read_prob[i]*rRS[i,j] + (1-read_prob[i])*wRS[i,j];
86  param target_model_rc_min_limit{i in 1..N, j in 1..M} :=
87              min(Q[i], U[j]/target_avg_rs[i,j]);
88  var rc{i in 1..N, j in 1..M} =
89              ceil(max(target_model_rc_min_limit[i,j], Q_ij[i,j]/D[j]));
90
91  # Read/Write Service Time: Look-Up in the performance table
92  var readIO_time{i in 1..N,j in 1..M} =
93                  tableLookUp_r(rRS[i,j], rc[i,j], cf[i,j], T[j]);
94  var writeIO_time{i in 1..N,j in 1..M} =
95                  tableLookUp_w(wRS[i,j], rc[i,j], cf[i,j], T[j]);
96
97  #
98  # OBJECTIVE (min-max target utilization)
99  #
100 minimize Objective: max_util;
101
102 #
103 # CONSTRAINTS
104 #
105 subject to
106
107 # 1. Integrity Constraint
108 integrity{i in 1..N}:   (sum{j in 1..M} conf_matrix[i,j]) = 1;
109
110 # 2. Capacity Constraint
111 capacity{j in 1..M}:    (sum{i in 1..N} conf_matrix[i,j] * s[i]) <= c[j];
112
113 # 3. Load Constraint
114 target_util{j in 1..M}: max_util >=
115         sum{i in 1..N} conf_matrix[i,j]*
116                     (rRR[i,j]*readIO_time[i,j] + wRR[i,j]*writeIO_time[i,j]);
```

Lines between 14 and 30 declare the object, workload and target parameters which were introduced in Figures 3.3, 3.5 and 3.9. All of the input parameters are represented as vectors (e.g., arrays) whose sizes are bound to either number of objects N or number of targets M. For example, at line 18, capacities of targets are represented as a vector of size M; and, at line 24, sizes of objects are held as a vector of size N. A layout problem *instance* has to provide these input parameters with values before the solver attempts to solve the problem.

The given AMPL model is a simplified instance of the generic model. At line 21, it is seen that each target can only have a single device type. Therefore, the given AMPL model assumes that any given target is always homogeneous; that is, it is made up of a single device *type*. This is not an unrealistic assumption though, because targets are always homogeneous by design. Performance of a target is bound to the performance of the slowest storage device in the target; as a result, the same or performance-wise similar storage devices are normally used to construct a target. Another simplification made in the presented AMPL model is regarding

the RAID level used to construct targets. As you may notice, this model does not include RAID levels of targets in the list of target parameters because it assumes that RAID0 is the only configuration. In our experimental evaluation, all RAID arrays were constructed using RAID0.

In the second part of the model (lines between 42 and 95), layout and target models are implemented. Workload transformations are based on the rules presented in Sections 3.2.2.1 and 3.2.2.2.2. The assumptions specified above simplifies the RAID transformations. Since we assume homogeneity within a target and and RAID0 as the only RAID level (i.e., no parity devices), it is enough to transform each I/O workload once because the utilization of a target is determined by the utilization of a single device in the target. At lines 92 and 94, we see how the AMPL code calls the black-box cost models. Looking-up a device's service time for a given workload is carried out by external functions.

In AMPL, there is a clear distinction between parameters (i.e., *param*) and variables (i.e., *var*). Simply, parameters are constant values; that is, their values do not change during the course of the solver's or regularizer's execution. On the other hand, variables may take different values at different times, depending on the state of the *decision variable* of the model. At line 71, we see the declaration of the layout matrix L, which is the decision variable. This matrix holds a valid layout at any time. During the course of the execution of the solver and the regularizer, different valid layouts are visited by changing the state of this matrix.

Lastly, in the third part (lines between 100 and 116), the objective of the layout optimization problem is stated. The objective is minimizing the maximum target utilization, and the calculation of the target utilizations is given at line 114.

References

[1] Gagan Aggarwal, Tomás Feder, Rajeev Motwani, Rina Panigrahy, and An Zhu. Algorithms for the database layout problem. In *Proc. International Conference on Database Theory (ICDT)*, pages 189–203, 2005.

[2] Rakesh Agrawal, Surajit Chaudhuri, Abhinandan Das, and Vivek R. Narasayya. Automating layout of relational databases. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 607–618, March 2003.

[3] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollór, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for Microsoft SQL server. In *International Conference on Very Large Data Bases (VLDB)*, pages 1110–1121, 2004.

[4] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic physical design tuning: workload as a sequence. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 683–694, 2006.

[5] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19:483–518, 2001.

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

[7] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair C. Veitch. Hippodrome: Running circles around storage administration. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 175–188, 2002.

[8] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Ergastulum: Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, 2005.

[9] 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.

[10] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc., 3rd edition, 2005.

[11] John S. Bucy and Gregory R. Ganger. The disksim simulation environment version 3.0 reference manual. Technical Report CMU–CS–03–102, Department of Computer Science Carnegie-Mellon University, 2003.

[12] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. Compressing SQL workloads. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, pages 488–499, 2002.

[13] Surajit Chaudhuri and Vivek R. Narasayya. Autoadmin 'What-if' index analysis utility. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 367–378, 1998.

[14] Huang-Jen Chen and Thomas D. C. Little. Physical storage organizations for time-dependent multimedia data. In *Proc. Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 19–34, 1993.

[15] Transaction Processing Council. Decision Support System Benchmark, http://www.tpc.org/tpch.

[16] Transaction Processing Council. Transaction Processing Benchmark, http://www.tpc.org/tpcc.

[17] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. Flexvol: flexible, efficient file volume virtualization in WAFL. In *Proc. USENIX Annual Technical Conference*, pages 129–142, 2008.

[18] EMC Corp. EMC CLARiiON MetaLUNs concepts, operations, and management. Engineering White Paper, Part Number H1024. http://canada.emc.com/collateral/hardware/white-papers/h1024-clariion-metaluns-cncpt-wp-ldv.pdf, October 2003.

[19] EMC Corp. EMC DMX-4 for Oracle 10g and Oracle 11g data warehouse layout. White Paper, Part Number 4005. http://www.emc.com/collateral/hardware/white-papers/h4005-symmetrix-dmx4-oracle-wp.pdf, October 2007.

[20] EMC Corp. EMC symmetrix DMX-4 series. http://www.emc.com/products/series/symmetrix-dmx-4.htm, Feb 2009.

[21] EMC Corp. Leveraging EMC CLARiiON CX4 with enterprise flash drives for Oracle database deployments. Engineering White Paper, Part Number h5967.3, October 2009.

[22] Ian Foster and Steven Tuecke. Describing the elephant: The different faces of IT as service. *ACM Queue*, 3(6):26–29, 2005.

[23] Robert Fourer, David M. Gay, and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Thomson Brooks/Cole, 2nd edition, 2003.

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

[25] Juraj Hromkovic. *Algorithms for Hard Problems*. Springer, 2nd edition, 2003.

[26] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 439–450, 1994.

[27] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proc. of File and Storage Technologies (FAST)*, pages 7–12, March-April 2004.

[28] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[29] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 2nd edition, 1973.

[30] Edward K. Lee and Randy H. Katz. An analytic performance model of disk arrays. *SIGMETRICS Perform. Eval. Rev.*, 21(1):98–109, 1993.

[31] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 1075–1086, 2008.

[32] Yung-Cheng Ma, Jih-Ching Chiu, Tien-Fu Chen, and Chung-Ping Chung. Variable-size data item placement for load and storage balancing. *Journal of Systems and Software*, 66(2):157–166, 2003.

[33] Patriot Memory. Warp Series SSD (Patriot PE32GS25SSDR). http://patriotmemory.com/products/eolp.jsp?prodline=8&catid=21&prodgroupid=83.

[34] Arif Merchant and Guillermo A. Alvarez. Disk array models in minerva. Technical Report HPL-2001- 118, HP Labs, 2001.

[35] Michael Mesnier, Matthew Wachs, Brandon Salmon, and Gregory Ganger. Relative fitness models for storage. *SIGMETRICS Performance Evaluation Review*, 33(4), 2006.

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

[37] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 239–248, 2005.

[38] NetApp Inc. Data ONTAP 7.3 Storage Management Guide. http://now.netapp.com/NOW/knowledge/docs/ontap/rel731/pdfs/ontap/smg.pdf, June 2008.

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

[40] Open Source Development Labs. Database Test 2 (OLTP) and Database Test 3 (DSS). http://osdldbt.sourceforge.net/.

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

[42] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-aware storage layout for database systems. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, 2010.

[43] Oguzhan Ozmen, Kenneth Salem, Mustafa Uysal, and M. Hossein Sheikh Attar. Storage workload estimation for database management systems. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 377–388, 2007.

[44] Jun Rao, Chun Zhang, Guy M. Lohman, and Nimrod Megiddo. Automating physical database design in a parallel database. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 558–569, 2002.

[45] Doron Rotem, Gerhard A. Schloss, and Arie Segev. Data allocation for multidisk databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):882–887, 1993.

[46] Juan Rubio, Charles Lefurgy, and Lizy Kurian 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.

[47] Aamer Sachedina, Matthew Huras, and Agatha Colangelo. Best practices database storage. White paper, IBM DB2 for Linux, UNIX, and Windows, Oct. 2008.

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

[49] S. Singhal, M. Arlitt, D. Beyer, S. Graupner, V. Machiraju, J. Pruyne, J. Rolia, A. Sahai, C. Santos, J. Ward, and X. Zhu. Quartermaster – a resource utility system. In *Proc. of the 9th IFIP/IEEE Intl. Symposium on Integrated Network Management*, May 2005.

[50] Sun Microsystems. *Sun Grid Compute Utility: Reference Guide*, June 2006. Part No. 819-5131-11.

[51] Seagate Techonology. Lowest cost-per-performance disc drive (Cheetah 15K RPM ST318453LC). http://www.seagate.com/docs/pdf/datasheet/ disc/ds_cheetah15k3_1552003_2003_03.pdf, March 2003.

[52] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. International Workshop on Modeling, Analysis, and Simulation of Computer and Telecom. Systems (MASCOTS)*, pages 183–192. IEEE, Aug. 2001.

[53] Elizabeth Varki, Arif Merchant, Jianzhang Xu, and Xiaozhou Qiu. An integrated performance model of disk arrays. In *Proc. International Symposium on Modeling, Analysis, and Simulation of Computer and Telecom. Systems (MASCOTS)*, pages 296–305, 2003.

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

[55] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory Ganger. Storage device performance prediction with CART models. In *Proc. ACM SIGMETRICS*, pages 412–413, 2004.

[56] Julie Ward, Michael O'Sullivan, Troy Shahoumian, and John Wilkes. Appia: automatic storage area network design. In *Conference on File and Storage Technology (FAST)*, pages 203–217, January 2002.

[57] Ted J. Wasserman, Patrick Martin, David B. Skillicorn, and Haider Rizvi. Developing a characterization of business intelligence workloads for sizing new database systems. In *Proc. of the 7th ACM International Workshop on Data Warehousing and OLAP*, pages 7–13. ACM Press, 2004.

[58] J. Wilkes, G. Janakiraman, P. Goldsack, L. Russell, S. Singhal, and A. Thomas. Eos – the dawn of the resource economy. In *8th Workshop on Hot Topics in Operating Systems*, May 2001.

[59] John Wilkes. The pantheon storage-system simulator. Technical Report HPLSSP9514, HP Labs, 1996.

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

[61] John Wilkes. Traveling to Rome: a retrospective on the journey. *SIGOPS Operating Systems Rev.*, 43(1):10–15, 2009.

[62] Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M. Lohman, Roberta Cochrane, Hamid Pirahesh, Latha S. Colby, Jarek Gryz, Eric Alton, Dongming Liang, and Gary Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *IEEE International Conference on Autonomic Computing*, pages 180–188, 2004.