Storage Workload Estimation for Database Management Systems

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

Mustafa Uysal
Hewlett Packard Laboratories
Palo Alto, CA USA
mustafa.uysal@hp.com

M. Hossein Sheikh Attar
University of Waterloo
Waterloo, ON Canada

ABSTRACT

Modern storage systems are sophisticated. Simple direct-attached storage devices are giving way to storage systems that are shared, flexible, virtualized and network-attached. Today, storage systems have their own administrators, who use specialized tools and expertise to configure and manage storage resources. Although the separation of storage management and database management has many advantages, it also introduces problems. 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. In this paper, we attempt to close this gap by addressing 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. Such a characterization can be used by a storage administrator to guide storage configuration. The ultimate goal of this work is to enable effective end-to-end design and configuration spanning both the database and storage system tiers. We present an empirical assessment of the cost of workload prediction as well as the accuracy of the result.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems; D.4.2 [Operating Systems]: Storage Management

General Terms: Algorithms, Experimentation, Management, Performance

Keywords: workload characterization, storage management, storage configuration, database management systems

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 [6, 9, 12, 13, 19]. In the case of storage resources, simple, direct-attached storage devices are giving way to shared, flexible, virtualized, network-attached 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, storage resources are increasingly administered separately from the server infrastructure; storage administrators are expected to balance the requirements of multiple database systems and other storage clients. 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 (SA) 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 bottlenecks or resource shortages. Configuration decisions made by the SA determine the performance, reliability, and capacity characteristics of the storage system as seen by the DBMS. To help SAs cope with the complexity of these tasks, researchers have developed storage management tools that can be used to automate storage design and configuration tasks [3, 4, 8, 16].

Effective storage administration, whether manual or automatic, depends on knowledge of the storage system workload. However, accurate workload characterizations can be difficult to come by, particularly at initial configuration time. Often storage administrators must rely on rough workload "guesstimates", perhaps informed by previous experience with other systems or general knowledge of the clients that the storage system is expected to support. 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 this paper, we attempt to close the information gap between the database tier and the storage tier by address-

SIGMOD'07 June 11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

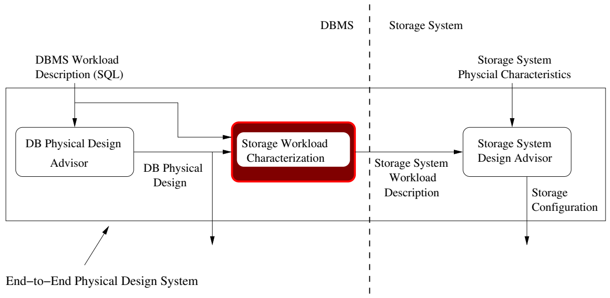

DBMS
Storage System
DBMS Workload Description (SQL)
Storage System Physcial Characteristics
DB Physical Design Advisor
Storage Workload Characterization
Storage System Design Advisor
DB Physical Design
Storage System Workload Description
Storage Configuration
End–to–End Physical Design System

Figure 1: End-to-End Physical Design using Existing Design Advisers

ing 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. 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. In doing so, we enable end-to-end solutions to physical design and storage configuration problems. One example of this is shown in Figure 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. 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.

This paper 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 [18]. By "domain-independent", we mean that the workload description that is produced is not specific to database management systems. Similar descriptions can be produced by other storage system clients. As storage consolidation becomes more common, this property becomes more important.
- We present a technique for producing storage workload estimates. Our technique has been implemented in the context of the Postgres DBMS.
- We present an empirical evaluation of the accuracy of the storage workload estimates produced by our technique, and the cost of producing them.

The remainder of this paper is structured as follows. Section 2 defines the input database workload, the target storage workload model and the workload estimation problem. Section 3 presents our workload estimation technique and its implementation in Postgres, and Section 4 describes its evaluation. Section 5 provides a brief survey of related work, and Section 6 concludes.

2. PROBLEM FORMULATION

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, we begin by defining what we mean by "database workload" and "storage workload".

2.1 Database Workload Model

Existing relational database design tools typically expect the database workload to be defined as a set of SQL statements (queries and updates) along with some indication of the relative frequency of occurrence of each statement [2, 20]. We use a similar characterization of the database workload for our storage workload estimation problem, so that a single workload description can be used for both tasks. Specifically, we assume 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 Q_i has an associated weight f_i which represents its prevalence in the workload. The proportion of queries of type 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 and updates 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, we also require 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 by λ. The second is the query multiprogramming level, k, which describes the expected number of concurrently executing queries at any given time.

Finally, since our storage workload estimator relies on the database system's query optimizer, we require 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 [5].

We assume that a database physical design has been selected, perhaps through the use of a physical design advisor [2, 20], 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, materialized views and so on. Figure 2 summarizes the database workload parameters.

2.2 I/O 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 populat-

Symbol	Description
$\mathcal{Q}$	set of possible SQL statements (query types)
f_i	relative frequency of query type Q_i
λ	query throughput
k	number of concurrent queries
$\mathcal{D}$	set of database physical objects

Figure 2: Database Workload Model Parameters

ing 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 [18]. 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 [3, 4, 16]. 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 any kind of storage client. Since shared, consolidated storage systems must accommodate workloads from a variety of clients, including databases, we believe that 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, we are also able 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. The starting position of each request is determined by a *run length* parameter L. Successive requests in a stream start where the previous request left off, until the total number of requests in the run reaches L. The next request then starts a new run, with a randomly chosen starting position. Thus, $L = 1$ models a random I/O request pattern, while larger values of L model sequentiality. Figure 3 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* $\mathcal{C}$. Entry $\mathcal{C}[i, j]$ in the overlap matrix 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 $\mathcal{C}[i, j] = 10\%$ and $\mathcal{C}[j, i] = 100\%$.

Symbol	Description
t_{on}	burst duration
t_{off}	inter-burst gap
λ_r	read request rate during bursts
λ_w	write request rate during bursts
B	size of each request
L	total length of a sequential run
$\mathcal{C}[i, j]$	burst overlap between streams i and j

Figure 3: I/O Request Stream Parameters in Rome

2.3 The Storage Workload Estimation Problem

With the definitions of a database workload and storage workload in place, we can now state our problem:

DEFINITION 2.1. ***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.*

For now, we will leave open the issue of how to measure the accuracy of the resulting storage workload model. We will address this issue in Section 4.

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?

In this paper, we have simplified the workload estimation problem 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, we restrict our attention to workload models that include exactly $|\mathcal{D}|$ Rome stores, one for each physical database object. 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, we restrict our attention to workload models with a single request stream per store. 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 here because they are simpler. However, the storage estimation method described in Section 3 can easily be extended

```
1  foreach query type $Q_i \in \mathcal{Q}$
2    Use the database query optimizer to obtain a plan for $Q_i$
3    Generate an I/O request sequence $S_i$ for each $Q_i$'s plan.
4  end
5  Merge the $S_i$ to produce a representative I/O request trace $T$ for the DB workload
6  foreach physical database object $D_j \in \mathcal{D}$
7    Extract the representative request trace $T_j$ for $D_j$ from $T$
8    fit Rome model parameters to $T_j$
9  end
```

Figure 4: High-Level Description of Storage Workload Estimation Method

to generate $|\mathcal{Q}|$-streams-per-model if additional expressiveness is required. Furthermore, existing Rome-based storage management tools can accommodate multi-stream stores.

3. WORKLOAD ESTIMATION

Figure 4 gives a high-level outline of our method of estimating a Rome I/O workload model. As described in Section 2.3, the output of this method is one set of Rome I/O model parameter values (as shown in Table 3) for each physical database object $D_j \in \mathcal{D}$. The model parameters for 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 4 consists of three phases. First, we generate an I/O request sequence corresponding to each query type in the database workload (Figure 4 lines 1-4). Second, we merge those individual sequences into a single I/O request trace, which we call the *representative I/O trace* for the given database workload and operating point (line 5). Finally, we project each physical object's requests from the representative trace and fit the Rome stream parameters to the projected trace (lines 6-9). In the remainder of this section, we describe each of these phases in more detail.

3.1 Estimating Query Request Sequences

An *I/O request sequence* is an ordered list of records, each of which describes a single I/O operation. Specifically, each record consists of the following fields: physical object identifier, starting offset within the physical object, request length, and request type (read or write). Note that, in Figure 4, we have distinguished request sequences from *request traces*. 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 5 summarizes our approach.

To obtain these sequences, we perform a *data-free simulation* of the control flow of each query's execution plan. 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. 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 our data-free simulation, we 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 optimizer assumptions. For example, a sort operation is assumed to create initial runs that are twice the size of the working memory available for the sort.

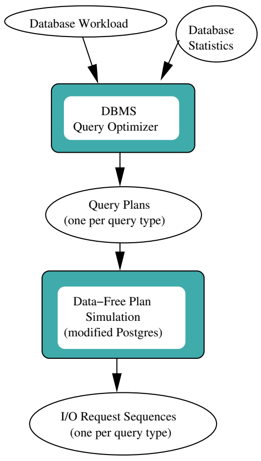
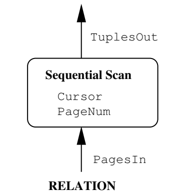
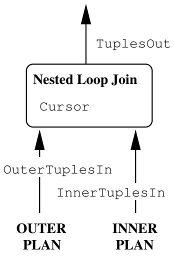

Database Workload
Database Statistics
DBMS Query Optimizer
Query Plans (one per query type)
Data–Free Plan Simulation (modified Postgres)
I/O Request Sequences (one per query type)

Figure 5: Generating I/O Request Sequences

By performing the data-free simulations, we hope 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 our own 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, as well as information from the database catalogue. For example, a table scan of a relation will generate sequential requests, while an index scan of the same relation using an uncorrelated 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

Operator	Handling Init()	Handling getNext()

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

 TuplesOut
Nested Loop Join
Cursor
OuterTuplesIn
InnerTuplesIn
OUTER PLAN
INNER PLAN
	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);

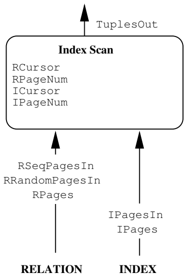
 TuplesOut
Index Scan
RCursor
RPageNum
ICursor
IPageNum
RSeqPagesIn
RRandomPagesIn
RPages
IPagesIn
IPages
RELATION
INDEX
	RCursor := 0;
	RPageNum :=
	    random(0,RPages-RSeqPagesIn);
	ICursor :=
	    random(0,IPages-IPagesIn);
	IPageNum := ICursor;
		Iposition := ceil(ICursor);
		ICursor += IPagesIn/TuplesOut;
		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);

Figure 6: Data-Free Simulation of Postgres 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 Postgres optimizer statistics and configuration parameters that are used by the simulator.

simulation understands that a hash join will first retrieve the entire build input and then retrieve the entire probe input, resulting to 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.

Our implementation of data-free simulation is embodied in a modified version of Postgres. In our version of Postgres, there are 18 different operators that may appear in execution plans. Our plan simulator handles most aspects of these operator types. One limitation of our current implementation is that certain kinds of SQL subqueries (those that result in query-valued qualifiers in plan nodes) are not handled. This is a restriction of our current prototype, not a fundamental restriction. We do not have space here to present the entire simulator. However, Figure 6 illustrates the simulation for three of the Postgres operators: sequential scan, index scan, and nested loop join.

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 4.4.

3.2 Generating the Representative Trace

The I/O request sequences generated in the first phase capture the I/O workload characteristics of a single workload query running in isolation. In the second phase, we generate 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 representative I/O trace 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 the database system's buffer cache 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

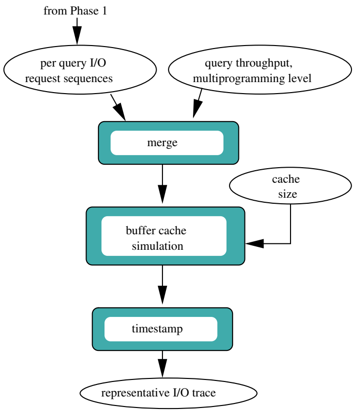


Figure 7: Generating the Representative I/O Trace

database system at the specified operating point.

Figure 7 summarizes the process of generating the representative I/O trace. We use a simple probabilistic operational model of the database system 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 k at the target operating point. k is specified as a workload parameter (see Figure 2). To generate a merged I/O sequence, k 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, we apply it to a DBMS-specific buffer cache model. To model the buffer cache, we are currently using a simulation of the 2Q cache replacement algorithm [7] that is used by Postgres. This simulation is parameterized 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.

Finally, we associate timing information with each remaining I/O request in the sequence to produce the representative I/O trace. To do this, we use the query throughput λ that is supplied as a parameter to the workload estimation process. We first translate query throughput to I/O throughput by multiplying query throughput by the expected number of I/O requests per query:

$$\lambda_{io} = \lambda \frac{\sum_i N_i f_i}{\sum_i f_i}$$

where N_i is the cache-corrected length of I/O request sequence (from phase 1) for query type Q_i. The jth request in the representative I/O trace is assigned an arrival time of j/λ_{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.

3.3 Fitting the Rome Model

To produce a Rome model of the I/O workload, we must choose a set of Rome stream parameter values to characterize the I/O requests directed to each of the physical database objects. To select parameter values for a given object, we first project that object's requests from the aggregate representative trace, and then choose Rome parameter values (see Table 3) to fit the per-object trace.

We take advantage of an existing I/O trace analysis tool called Rubicon [15] 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 8 summarizes how each of the per-object Rome parameters is estimated. In addition to the parameters shown in Figure 8, Rubicon also estimates the burst overlap matrix, $\mathcal{C}$. Entry $\mathcal{C}[i,j]$ describes how stream j's bursts overlap with those of stream i. To estimate this, Rubicon measures the total amount of time during which both streams are simultaneously in I/O bursts. This is divided by the sum of stream i's burst durations to generate an estimate for $\mathcal{C}[i,j]$.

4. EVALUATION

In this section, we present an empirical evaluation of our storage workload estimation technique. Our evaluation has two goals. First, we would like to determine how accurate our storage workload estimates are. Second, we would like to determine how costly it is to generate those estimates.

What do we mean by accurate? One way to characterize accuracy is to compare, for a given database workload, the estimated storage 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 our case, our 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.

We have considered both types of evaluation. In Section 4.2, we present a direct comparison of estimated and measured workloads. In Section 4.3, we examine the utility of estimated workload traces for the purpose of predicting the performance of various storage system configurations. Section 4.4 presents measurements of the cost of estimation.

4.1 Experimental Configuration

We modified Postgres-8.0.6 so that it would produce I/O request logs during query execution. The request logs include one record for each I/O operation issued by Postgres. These logs capture the actual storage workload generated by the database system, against which we can compare our estimated workloads.

Symbol	Description	Estimation
t_{on}	burst duration	Trace requests are partitioned into bursts, with a request inter-arrival gap greater than 2 seconds indicating a burst boundary. t_{on} is estimated as the average duration of the resulting bursts.
t_{off}	inter-burst gap duration	Estimated as the average duration of the inter-burst gaps.
λ_r	read request rate during bursts	Estimated as the total number of read requests in the trace divided by the sum of the burst lengths.
λ_w	write request rate during bursts	Estimated as the total number of write requests in the trace divided by the sum of the burst lengths.
B	request size	Estimated as the average size of the requests in the trace.
L	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. L is estimated as the average length of the runs in the trace.

Figure 8: Estimation of Rome I/O Model Parameters using Rubicon Trace Analyzers

Postgres was running on a Dell Poweredge 2600 server with two 2.2 GHz Intel Xeon processors and 4 gigabytes of main memory, running SUSE 10.0 Linux with a 2.6.13-15.8 kernel. The server has a 70 gigabyte 15K RPM SCSI disk that is used to hold all system software, including Postgres itself, as well as the I/O logs generated by our modifications. In addition, the server has five 18.4 gigabyte 15K RPM hard drives behind a configurable Dell Perc 4di RAID controller. These drives were configured in a variety of ways, depending on the particular experiment that was running. Our default configuration grouped 4 of the disks into a single RAID0 logical device, on which we built a Reiserfs file system to hold the database. The fifth disk was used for transaction logging.

Postgres uses an 8 kilobyte page size, and we configured our system with a 2 gigabyte shared buffer. In addition, we minimized the impact of the Linux kernel's I/O buffering by allocating (non-pageable) huge pages that consumed the remaining main memory. The Postgres `work_mem` parameter, which controls the amount of memory used by operators that hash or sort, was set to 1 megabyte.

We experimented with instances of the TPC-H database. The default database workload used for many of our experiments used a TPC-H instance at scale factor 20. The queries consisted of randomly-generated instances of TPC-H query types Q1, Q3, Q5, Q6, Q10, Q12 and Q14, with each query type having equal probability of occurrence. The execution plans for these queries make use of a total of eleven physical database objects, including 7 tables (lineitem, orders, supplier, region, nation, customer, and part), 3 primary key indexes (orders_pkey, customer_pkey, and part_pkey) and an index (i_l_orderkey) on the orderkey attribute of the lineitem time. We will denote this workload by WSEQ20, as it generates primarily sequential I/O.

We also experimented with a mixed workload, WMIX5, consisting of the same TPC-H queries plus additional queries that generate a more random I/O workload. WMIX5 used a database instance at scale factor 5. WMIX5 is described further in Section 4.2.1.

4.2 Accuracy of Estimated Workloads

Our first goal was to directly compare estimated storage workloads with actual storage workloads generated by Postgres. Figure 9 illustrates the design of our experiment. Using our default WSEQ20 query workload at a specified multiprogramming level k, we generated queries and submitted them to Postgres for execution. We captured a trace of the actual storage workload generated by Postgres as it executed the queries, and also measured the query throughput, λ. We then used this workload, including the measured query throughput, as input to the storage workload estimator.[1] This produces an estimated storage workload model, M_{est}, as well as a representative I/O trace as an intermediate result. Finally, we fit a Rome workload model to the actual storage workload trace using the same Rubicon-based model fitting procedure used in the third phase of our estimation process. This results in a Rome model of the measured storage workload, which we denote by M_{meas}.

Figure 10 gives a direct comparison of the I/O request counts found in the actual workload trace and the estimated representative trace. The columns labeled WSEQ20 show this comparison for the WSEQ20 workload at two concurrency levels, $k = 1$ and $k = 5$. The columns labeled WMIX5 are discussed in Section 4.2.1. With no database concurrency ($k = 1$), the estimated I/O request counts are very accurate. This reflects the fact that the Postgres optimizer estimates used by the storage estimator are quite accurate for the WSEQ20 workload. Under these workload conditions the Postgres buffer cache is largely ineffective, and this is captured by the estimator's cache simulation.

Introducing concurrency into the database workload ($k = 5$) introduces some error into the I/O request counts: the total count is overestimated by about 15%. Since the isolated per-query request traces have accurate I/O counts, as shown by the $k = 1$ case, we attribute this error largely to the cache simulation. Although the simulator itself mimics the Postgres buffer manager, the request sequence seen by the cache will be different in the estimator than in reality, and thus the simulated buffer cache does not, in general, have the same performance as the actual buffer cache.

[1] We ensured that the estimator ran the same set of queries as Postgres, and that they were initiated in the same order as they were in Postgres. This ensures that any storage modeling error can be attributed to our methodology and not to differences in the database workloads seen by Postgres and the estimator.

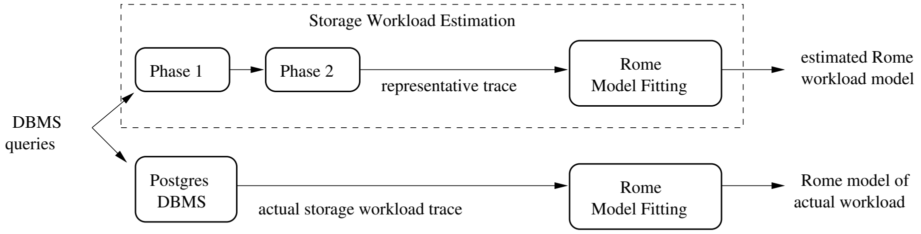


Figure 9: Experiment to Compare Estimated and Measured Storage Workloads

	WSEQ20 @ $k = 1$			WSEQ20 @ $k = 5$			WMIX5 @ $k = 5$		
object	measured	estimated	error	measured	estimated	error	measured	estimated	error
lineitem	20891772	20891784	0.0%	62363598	74365812	19.2%	9637369	7767389	-19.4%
orders_pkey	273564	274221	0.2%	1002063	1004521	0.2%	22799	22864	0.3%
orders	2928151	2927070	0.0%	11596208	11585626	0.0%	585604	531263	-9.3%
supplier	4919	4919	0.0%	19676	19676	0.0%	0	0	0.0%
i_l_orderkey	786003	788223	0.3%	4451114	4702823	5.7%	262020	262754	0.3%
region	1	1	0.0%	4	4	0.0%	0	0	0.0%
nation	2	2	0.0%	11	11	0.0%	1	1	0.0%
customer	241566	241558	0.0%	1256966	1071097	-14.8%	60415	80551	33.3%
customer_pkey	9122	9149	0.3%	63856	73192	14.6%	2283	2289	0.3%
part_pkey	12161	12197	0.3%	48644	48788	0.3%	9126	9150	0.3%
part	97251	97251	0.0%	389272	389004	0.0%	72942	72942	0.0%
i_l_commitdate	0	0	0.0%	0	0	0.0%	11508	8102	-29.6%
TOTAL	25244512	25246375	0.0%	81191412	93260554	14.9%	10664067	8757305	-17.9%

Figure 10: Estimated vs. Measured I/O Counts

I/O request counts are only one property of the I/O workload. As described in Section 2.2, our I/O workload model captures other significant properties, including request rates, burstiness, and sequentiality. Figure 11 compares some of these statistics in measured (M_{meas}) and estimated (M_{est}) workload models.[2] Each bubble in these graphs represents one database object (table or index), with bubble sizes scaled to the object's actual I/O request count. Thus, more important objects have larger bubbles. Figure 12 summarizes these results as weighted average estimation errors over all of the objects' request streams.

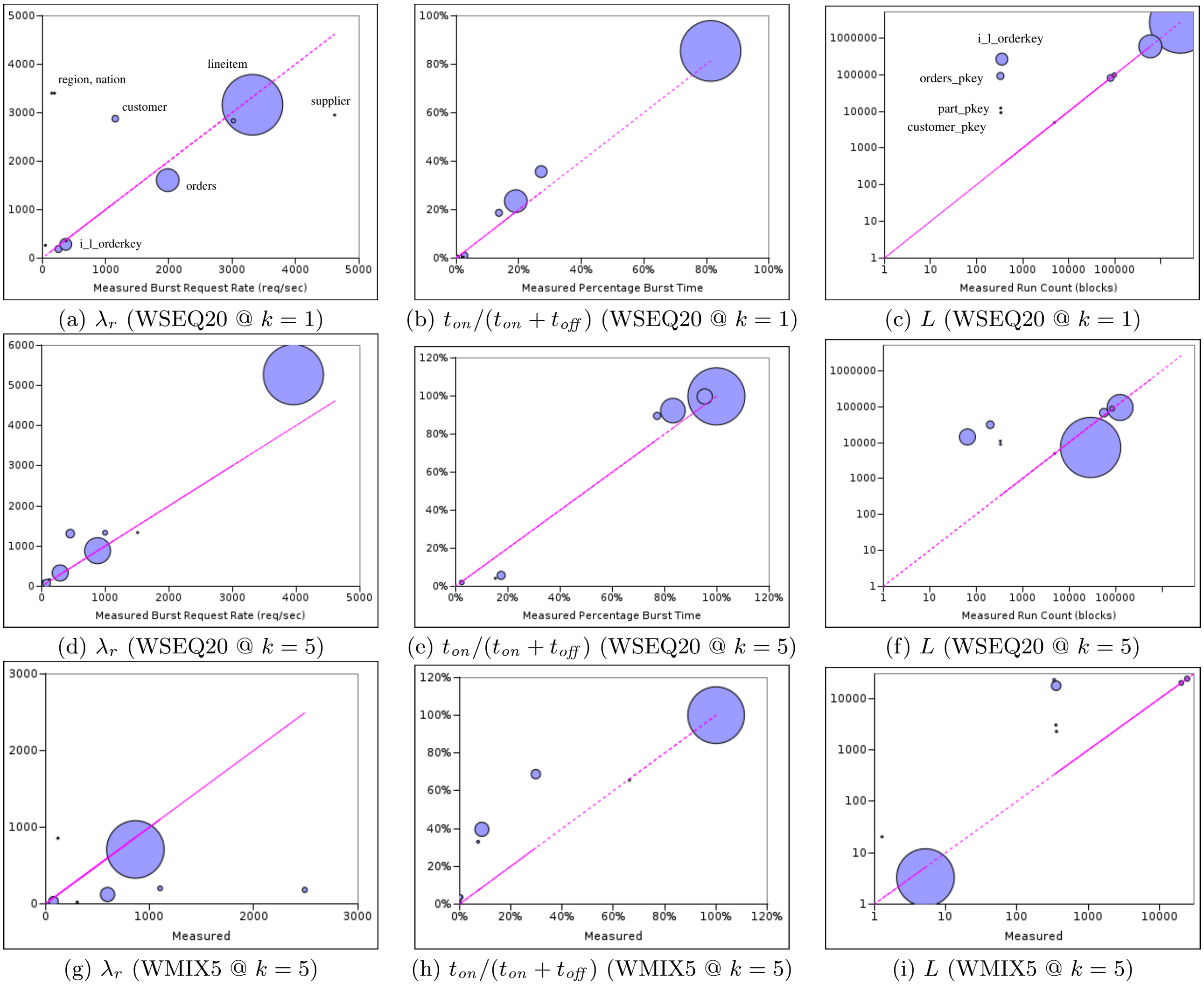
Burst Request Rate: With no concurrency, burst rate estimates are fairly accurate. The accuracy decreases somewhat with the addition of concurrency. Figure 11(d) shows that this increase can be attributed largely to overestimation of lineitem's burst rate when $k = 5$. This, in turn, is a largely a reflection of the overestimation of the total I/O count for lineitem, as was shown in Figure 10. There are a few outliers in Figure 11(d), but these are tiny tables like nation and region that make a negligible contribution to the total I/O workload. The estimation errors for those tables are artifacts of the way that the lengths of very short bursts are determined by Rubicon.

Percentage Burst Time: Low burst time percentages indicate bursty I/O streams with relatively long idle times, while high burst time percentages indicate smoother streams. Not surprisingly, increasing the query concurrency makes most of the I/O request streams smoother, as can be seen by comparing Figures 11(b) and 11(e). The estimator is accurate with and without concurrency (Figure 12).

Run Length: The estimator makes substantial errors in predicting run lengths (Figure 11), largely due to errors of several orders of magnitude in the run count predictions of index objects (Figure 11(c) and 11(f)). These errors can be traced to the index scan simulation, which makes a simplifying assumption that an index scan results in sequential access to all index pages. In reality, some non-leaf pages are never accessed and not all pages that are accessed are accessed sequentially because of the way that index pages are laid out. Fortunately, this kind of run length estimation error is probably not significant. *Any* L 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., $L = 1$) and large run lengths. In Section 4.3, 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 is not shown in Figures 11 and 12 is the burst overlap matrix, $\mathcal{C}$. Figure 13 compares measured and estimated burst overlaps, $\mathcal{C}[i, j]$, among the database objects' request streams for the case of $k = 5$. Each graph shows the percentage overlap of one object's request bursts with the bursts of other objects. The nation, region, supplier objects are not included as they make a negligible contribution to the workload. These data show

[2] Two I/O workload model parameters, λ_w and B, are not represented in Figure 11. For our workload, $\lambda_w = 0$ and $B = 8192$ bytes. Both of these parameters are captured accurately by the estimator.



(a) λ_r (WSEQ20 @ $k = 1$) (b) $t_{on}/(t_{on} + t_{off})$ (WSEQ20 @ $k = 1$) (c) L (WSEQ20 @ $k = 1$)

(d) λ_r (WSEQ20 @ $k = 5$) (e) $t_{on}/(t_{on} + t_{off})$ (WSEQ20 @ $k = 5$) (f) L (WSEQ20 @ $k = 5$)

(g) λ_r (WMIX5 @ $k = 5$) (h) $t_{on}/(t_{on} + t_{off})$ (WMIX5 @ $k = 5$) (i) L (WMIX5 @ $k = 5$)

Figure 11: 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.

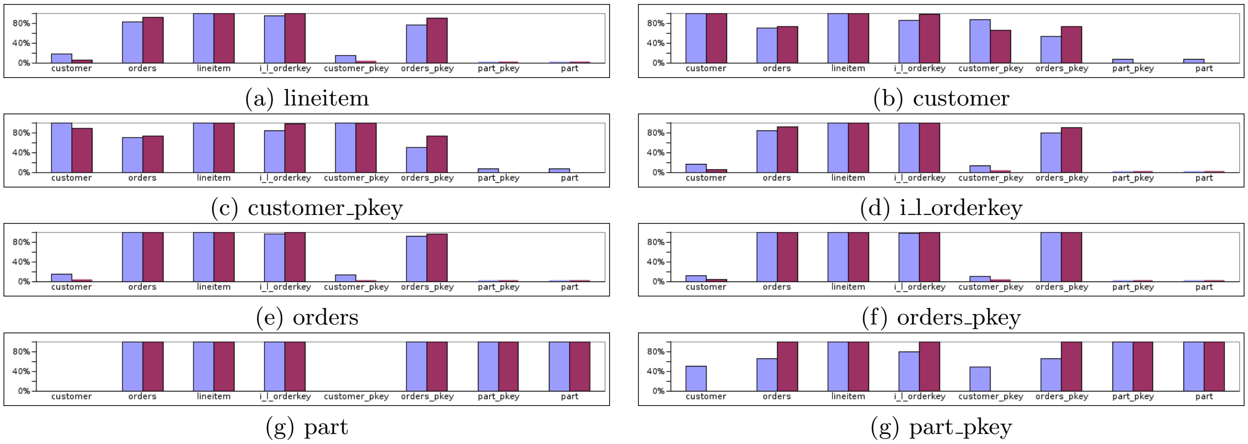
Workload	Burst Request Rate λ_r	Percentage Burst Time $\frac{t_{on}}{t_{on}+t_{off}}$	Run Length L
WSEQ20 @ $k = 1$	9%	9%	2602%
WSEQ20 @ $k = 5$	30%	3%	1450%
WMIX5 @ $k = 5$	23%	37%	174%

Figure 12: 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.

that the estimator does a very good job of estimating which objects' request bursts overlap and which do not. For example, when the orders table is being accessed (Figure 13(e)), so are the lineitem table, the orders table's primary index (orders_pkey), and the i_l_orderkey index. Other objects, like the customer table and its primary index, 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 [1].

4.2.1 *Accuracy Under a Mixed Workload*

We repeated the experiment of Figure 9 using a mixed workload, WMIX5. This workload used 5 concurrent streams of queries, i.e., $k = 5$. One stream consisted of the same TPC-H query mixture as WSEQ20. The four remaining streams ran simple selection queries against the lineitem table, using a range predicate on the commitdate attribute. These queries have a selectivity of 0.1%, and the Postgres optimizer selects an execution plan that uses a secondary in-



(a) lineitem

(b) customer

(c) customer_pkey

(d) i_l_orderkey

(e) orders

(f) orders_pkey

(g) part

(g) part_pkey

Figure 13: Measured and Estimated Burst Overlaps, $\mathcal{C}[i, j]$. The left bar of each pair shows measured values, the right bar shows estimated values.

dex on commitdate. Specific range predicates for each query are randomly selected from 1000 non-overlapping possibilities with uniform selection probabilities. The results are shown in Figures 10, 11, and 12 in the rows or columns labeled "WMIX5".

The most significant difference between the WMIX5 workload and the WSEQ20 workload is the nature of the accesses to the lineitem table, which we expect to shift from sequential in WSEQ20 to largely random in WMIX5. This is, in fact, what happens, as can be seen by comparing Figures 11(f) and 11(i). For lineitem in WSEQ20 at $k = 5$, $L \approx 30000$. For lineitem in WMIX5 at $k = 5$, $L \approx 5$. The workload estimator correctly characterizes this shift.

Estimation errors under WMIX5 were of comparable magnitude to those under WSEQ20, although the nature of the errors differed. Total I/O count was underestimated for WMIX5 by about the same amount as it was overestimated under WSEQ20 (Figure 10). Again, we attribute this largely to error in caching simulation, especially for the lineitem table. Request rate prediction was more accurate under WMIX5, but burst time prediction was less accurate (Figure 12). For WMIX5, the estimated bursts were longer and less intense than the real bursts, i.e., the estimated request rate was smoother and less bursty than in reality. We are uncertain of the cause of this, but we suspect that it is related to the way that the individual query request patterns are merged to form the representative I/O trace in phase two of the estimation process.

4.3 Storage Performance Prediction Using Estimated Workloads

As was noted in the Section 1, 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 Disk Array Designer [4], use storage system models to estimate the performance of candidate configurations. In this section, we present the results of an experiment in which we used both measured and estimated storage workload models 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 estimated workload models were as good as the measured models for this particular task.

For this experiment, we used Delphi storage system performance models [14]. Delphi models are modular analytic models that take Rome storage workload descriptions as input. Thus, the output of our workload estimator can be used without modification as input to a Delphi model. A Delphi model of a storage system is composed of individual models of the components of the storage system, together with a description of how the components are interconnected. Components include things like individual disk drives, array controllers, caches, and so on. Each component predicts how it will perform in response to a given workload. In addition, components that pass workload "through" to other components predict how the workload will be transformed as it is passed through. For example, a RAID controller model describes how its input is divided among the disk drives that form a particular RAID group. In addition to the individual component and component interconnection models, a Delphi model describes how the "stores" that are referred to in the workload are mapped to storage system devices. In our case, each store corresponds to a physical database object, so these bindings effectively describe how database objects are laid out on the available storage devices.

Our Delphi models predict storage system utilization for a given workload. The utilization of the entire storage system is determined by the most heavily utilized storage system component. More precisely, since workloads are bursty, the models predict *peak utilization* over all burst periods, except that very short bursts are ignored. It would be relatively simple to change these models to predict, say, average utilization rather than peak utilization.

We have developed Delphi component models for the storage system components in our experimental environment (Section 4.1), which we can compose to build Delphi models of our storage system in various configurations, and for various data layouts. To run each experiment, we first chose a target configuration and configured our storage system accordingly. We then followed the procedure shown in Figure 9 to generate measured (M_{meas}) and estimated (M_{est}) storage workload model. We then fed each of these workload models into a Delphi model of our target storage system configuration and compared the storage system utilizations predicted

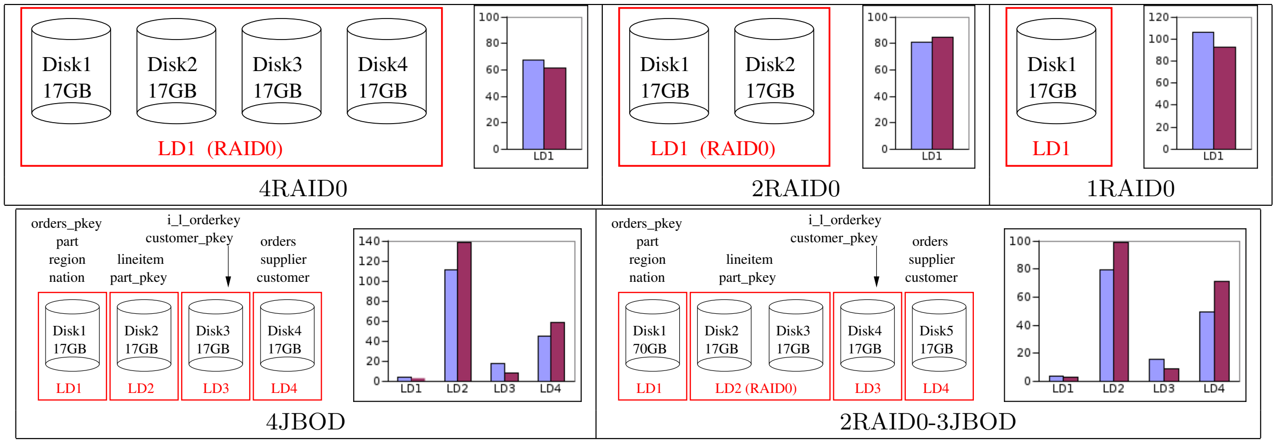


Figure 14: Storage System Performance Predictions Under Estimated and Measured Storage Workload Models. Each storage system configuration is illustrated next to a graph showing the measured and estimated peak percentage utilization of each logical device. In the graphs, the left bar in every pair shows utilization under M_{meas}, and the right bar shows utilization under M_{est}.

workload	Phase 1	Phase 2	Phase 3	Total
WSEQ20 @ $k = 1$	273	86	83	435
WMIX5 @ $k = 5$	116	121	35	293

Figure 15: Total and Per-Phase Times (seconds) for Storage Workload Estimation

by the Delphi model under each of the two workloads.

Figure 14 illustrates the storage system configurations that we tested, as well as the predicted storage system utilizations under M_{meas} and M_{est}. Three of the configurations (4RAID0, 2RAID0, 1RAID0) defined a single RAID0 logical device (labeled LD1 in Figure 14) for use by the DBMS. The other two configurations (4JBOD and 2RAID0-3JBOD) defined multiple logical devices. For those configurations, Figure 14 specifies which physical database objects were mapped to the each logical device.

Absolute errors in predicted utilization ranged as high as 25%, although they were often much lower. Note that the 4JBOD and 2RAID0-3JBOD 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 single workload and 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 (short of populating the database, running a workload, and measuring resulting the storage system load) are guesstimates and rules of thumb.

4.4 Cost of Storage Workload Estimation

Figure 15 shows the wall-clock time required for storage workload estimation under our two workloads. Recall that Phase 1 includes the generation of storage request sequences for individual queries. This takes longer for WSEQ20 than for WMIX5 because of the larger database size. In general, larger databases will result in longer per-query storage request sequences. This is one of the limitations of our trace-based approach. Phase 2 includes the request sequence merging and cache simulation, and Phase 3 is the statistical analysis by Rubicon. These times depend primarily on the total length of the representative storage workload trace that is generated. In comparison, actual (non-simulated) execution of the workloads took approximately two hours (WSEQ20) and three hours (WMIX5) in our test configuration. This is an order of magnitude longer than the time required by our data-free simulation approach.

5. RELATED WORK

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 [2, 20] and partitioning relations [2, 11]. 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 this paper.

Agrawal, Chaudhuri, Das, and Narasayya addressed the problem of automating the layout of relational databases on a given set of storage devices [1]. Internally, their solution uses an access graph to characterize the I/O resulting from a given database workload. The graph describes estimated number of I/Os to each DB object and edge weights that characterize co-access (similar to our overlap matrix $\mathcal{C}$ in our Rome-based descriptions). This is a less expressive model than the one we have used. For example, it makes no distinction between sequential and random I/O to an object and no distinction between reads and writes. More significantly, that work views storage layout as a a database administration problem. In contrast, our goal is to generate accurate database workload characterization to enable storage administrators to make informed decisions about layout and other related problems.

Wasserman, Martin, Skillcorn and Rizvi [17] describe a workload characterization approach for database systems. They characterize according to several resource-related attributes, such as CPU consumption and sequential and random I/O rates, as well as other properties such as join degree. 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.

Narayanan, Thereska and Ailamaki describe a database resource advisor for predicting transaction response times and throughput based on end-to-end tracing [10]. Their technique relies on instrumentation and tracing 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. This 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.

There are several tools that address the automation of storage system design and management, though these are somewhat less mature than production database physical design advisors. Disk Array Designer [4] addresses the problem of storage system configuration: which arrays to define, how to configure each array, and how to lay out application data to the arrays. Hippodrome [3] uses these design tools to automate the management of a storage system as the workloads change, using a measure, analyze, reconfigure cycle. Similar design and automation tools also exists for designing storage area networks [16] (SANs) that connect storage devices to servers, and for designing data reliability solutions (e.g., backups, mirrors, snapshots, etc) and configurations [8]. All of these storage layer tools require storage workload characterizations, and can directly take advantage of our storage workload estimator.

6. CONCLUSION

We have presented a technique for estimating the storage system workloads that are generated by database management systems. Our technique generates storage workload models in a form that is easily used by storage administration tools, such as configuration advisors. We have demonstrated the feasibility of this approach by implementing it in Postgres. Our experimental results suggest that the workload estimations produced by our 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.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, A. Das, and V. Narasayya. Automating layout of relational databases. In *International Conference on Data Engineering (ICDE'03)*, pages 607–618, 2003.

[2] S. Agrawal, S. Chaudhuri, L. Kollór, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL server. In *International Conference on Very Large Data Bases (VLDB '04)*, pages 1110–1121, 2004.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Conf. on File and Storage Technology*, pages 175–188, Jan. 2002.

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

[5] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 367–378, 1998.

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

[7] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. International Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, 1994.

[8] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proc. of File and Storage Technologies (FAST'04)*, pages 7–12, March-April 2004.

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

[10] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, pages 239–248, 2005.

[11] J. Rao, C. Zhang, G. M. Lohman, and N. Megiddo. Automating physical database design in a parallel database. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 558–569, 2002.

[12] 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 *Proceedings of the 9th IFIP/IEEE Intl. Symposium on Integrated Network Management*, May 2005.

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

[14] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proceedings of the Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2001)*, pages 183–192, 2001.

[15] A. Veitch and K. Keeton. The Rubicon workload characterization tool. Technical Report HPL-SSP-2003-13, HP Laboratories, Mar. 2003.

[16] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes. Appia: automatic storage area network design. In *Conference on File and Storage Technology (FAST'02)*, pages 203–217, Jan. 2002.

[17] T. J. Wasserman, P. Martin, D. B. Skillicorn, and H. Rizvi. Developing a characterization of business intelligence workloads for sizing new database systems. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, pages 7–13. ACM Press, 2004.

[18] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proc. Intl. Workshop on Quality of Service (IWQoS'2001)*, number 2092 in Lecture Notes in Computer Science, pages 75–91. Springer-Verlag, June 2001.

[19] 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.

[20] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. Cochrane, H. Pirahesh, L. S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *IEEE Int'l Conf. on Autonomic Computing*, pages 180–188, 2004.