Study of Different types of Storage and Memory Workload Estimation and Management for DBMS, RDBMS and DDBMS

NIRBHAI SINGH
Research Scholar, Manav Bharti University, H.P., India

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-toend 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.

----------♦----------

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 re-sources, 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 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.

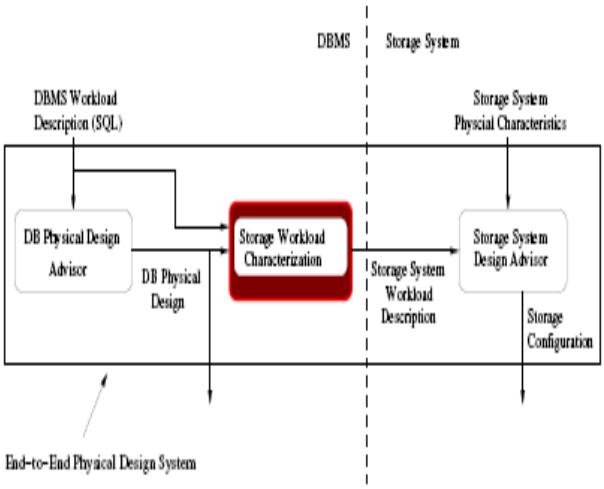


Figure 1: End-to-End Physical Design using Existing Design Advisers

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.

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 Q of SQL statements defined over a known database schema. We refer to each such statement as a query type. Each query type Qi has an associated weight fi which represents its prevalence in the workload. The proportion of queries of type Qi 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 _. 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 costbased "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 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-ing the database and applying a realistic load. Trace-based workload descriptions cannot be used as input to analytical models of storage system behavior.

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

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 ton 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|×|S| overlap matrix C. Entry C[i, j] in the overlap matrix describes the percentage of stream i's burst period during which stream j is also active.

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

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 ton [j] = 10, for which S_j 's bursts are completely contained within S_i's bursts. This will be described by C[i, j] = 10% and C[j, i] = 100%.

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 Dj 2 D. The model parameters for Dj 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.

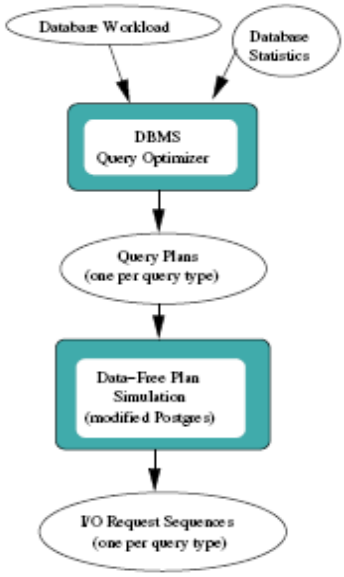
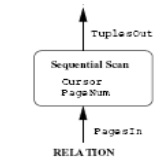
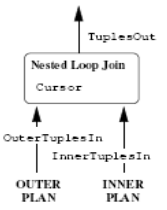


Figure 5: Generating I/O Request Sequences

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 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.

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);

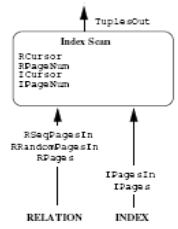
	RCursor := 0;
	RPageNum :=
	random(0,RPages-RSeqPagesIn);
	ICursor :=
	random(0,IPages-IPagesIn);
	IPageNum := ICursor;
		Iposition := ceil(ICursor);
		ICursor += IPagesIn/TuplesOut;
		Rposition := ceil(RCursor);
		RCursor += (RSeqPagesIn+RRandomPagesIn)/TuplesOut;
		for i := 1 to (ceil(ICursor)-Iposition)
		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.

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 perquery 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.

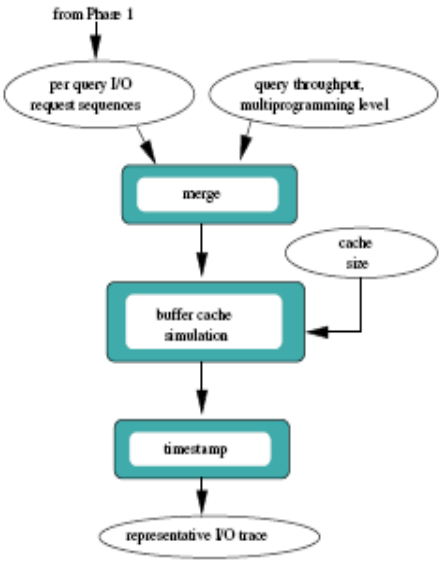


Figure 7: Generating the Representative I/O Trace

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 fi. 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 _ 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 Ni is the cache-corrected length of I/O request sequence (from phase 1) for query type Qi. The jth request in the representative I/O trace is assigned an arrival time of j/_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.

4. 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 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.

5. 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.

,

6. 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. Koll´or, 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.