LoadIQ: Learning to Identify Workload Phases from a Live Storage Trace

Pankaj Pipada, Achintya Kundu, K. Gopinath, Chiranjib Bhattacharyya
*Indian Institute of Science**

Sai Susarla, P. C. Nagesh
NetApp†

Abstract

Storage infrastructure in large-scale cloud data center environments must support applications with diverse, time-varying data access patterns while observing the quality of service. Deeper storage hierarchies induced by solid state and rotating media are enabling new storage management tradeoffs that do not apply uniformly to all application phases at all times. To meet service level requirements in such heterogeneous application phases, storage management needs to be *phase-aware* and *adaptive*, i.e., to identify specific storage access patterns of applications as they occur and customize their handling accordingly.

This paper presents LoadIQ, a novel, versatile, adaptive, application phase detector for networked (file and block) storage systems. In a live deployment, LoadIQ analyzes traces and emits phase labels learnt on the fly by using Support Vector Machines(SVM), a state of the art classifier. Such labels could be used to generate alerts or to trigger phase-specific system tuning. Our results show that LoadIQ is able to identify workload phases (such as in TPC-DS) with accuracy $> 93\%$.

1 Introduction

The efficiency of a storage system can be improved through adaptive storage management if access to high-level workload information is possible. For instance, it can trigger system optimizations suited for specific workloads or phases[2] as they occur. It can also be used to alert the administrator when rare anomalous workload behaviors recur and need attention for troubleshooting.

A storage administrator or management system can easily optimize storage resources for an application if it has the same behavior throughout its execution. But long running applications typically go through multiple distinct phases[6, 12] (for example, a computation phase followed by a checkpoint phase). Detecting these phase transitions within an application has been problematic.

In this paper, we focus on building a robust, real-time application phase identification facility for shared storage systems enabling various use cases. To support adaptive storage management in complex virtualized data centers, an application phase identification engine needs to have the following properties:

- Dependable: it should accurately discriminate among known classes of workload phases preferably with quantifiable confidence, as well as be able to flag unknown workload phases as such.
- Extensible: To be useful in dynamic environments, it should support augmenting new phase types as well as newly discovered criteria for discriminating among them. It should be easy to incorporate new expert knowledge as it becomes available.
- Automated: It should identify phases in near real-time to support online adaptation, where manual intervention is impractical.
- Robust: It should be robust against inevitable flux in real-world workload patterns due to variations in intensity, time spread and client-side or network environment.

A non-intrusive way to observe the workloads is to analyze the packet traces of networked storage protocols (such as NFS, CIFS, FCP, iSCSI, and HTTP). By detecting application phase transitions, we show that these traces can be a rich and dependable source of contextual information for storage systems to use for managing an application in its life cycle.

Packet traces of these protocols have various features such as access offsets, opcode sequences, etc. These features can be used to describe a workload. SVM is a widely used classification technique that relies on *kernel functions* to encode similarity between a pair of observations. These Kernel functions help in elegantly combining such multiple trace features. We adapt SVMs for

trace analysis and add iterative self-correction capability to handle untrained patterns so as to meet the requirements listed above (except quantifiable confidence).

A key contribution of this paper is *Online phase labeling with self-correction*. We demonstrate how to build a tool to track workload phase shifts in real-time (every minute) from a live trace feed while annotating it using the above methodology. This tool automatically improves its ability to recognize previously untrained workloads over time. This is essential for real-world deployability, as it is impractical to pre-learn all workload patterns out there.

The rest of the paper is organized as follows. Section 2 describes the related work in trace analysis and classification, placing LoadIQ in context to prior work. Section 3 summarizes the mathematical theory behind our SVM based classification methodology and discusses our *similarity* computation methods. Section 4 presents our results for detecting phases in applications. Finally, we conclude in Section 5.

2 Related Work

Techniques to optimize storage management for applications can be broadly classified as static vs. dynamic. Dynamic techniques react to dynamic system events and IO characteristics. For example, BORG[2] reorders block layout dynamically to convert temporal locality in data access into spatial locality for reducing disk head seeks to improve latency. Static techniques deploy recommended best practice storage configurations for specific application types determined offline by rigorous analysis and experience. Most production storage systems take this approach, while employing some dynamism based on heuristics.

Machine Learning techniques such as HMMs have been used in the past to to dynamically drive prefetching and caching decisions[8]. Also, there is a substantial body of work[7, 5, 1] where file system trace analysis has been attempted to get aggregate information about systems and to understand the usage patterns of storage over time. As we focus on extracting specific patterns present within an application, gross analysis of the system is not useful.

Reverse-engineering a network trace to discover the application that created it is difficult due to numerous non-deterministic factors[11, 4]. Even though specific heuristics could be employed to separate phases in a specific application, we construct a generic technique which can work for a variety of applications and is robust against variations in environment and configuration.

It has been shown that there is a strong correlation between high-level application context and the IO patterns generated[9, 13]. Our approach exploits this correlation in IO patterns hidden in traces to infer the application context at run time. Previous attempts at identifying workloads[11] use request type sequence for classification. This limits the applicability of the work in VM environments where most requests are reads and writes only. Other information in network traces such as offsets can be useful when analyzing traces in such environments (also in SAN and database workloads). Detecting phases within an application is also not attempted. In this paper, we use trace features such as offsets and apply state of art Machine Learning tools[10] to overcome previous limitations for detecting application phases.

3 Methodology

A network trace contains heterogeneous information such as opcodes, offsets, etc. In this paper, we use this information to identify various phases in the application using a *classification* paradigm. Support Vector machines (SVMs)[10] are currently the state of art classifiers which perform well on real world problems. Given current advances in multiple kernel learning, SVMs provide provide an excellent way of combining multiple sources of information. Even though we have used a single feature for analysis in this paper, the framework provided can be extended to use multiple trace features.

SVMs interact with data through *kernel functions*[10]. A kernel function is a *measure of similarity* between a pair of objects; here, these are NFS traces. In the next subsection we describe our *kernel* computation method. We describe how classification is done and then show how online self-correction is attempted.

3.1 Kernel computation using Offset Histograms

To use read/write offset fields of the NFS trace for classification we require a similarity measure computed on these fields. We extract offset fields from the NFS trace's `READ` and `WRITE` requests and compute a histogram out of the absolute difference between each successive offset fields (i.e, offset shift). We quantize the offset shifts into their nearest *bin* sizes in powers of 2, i.e., sizes of 2^1, 2^2, 2^3, ... bytes. After constructing the histogram from offset shifts, we normalize it to eliminate unwanted effects due to different trace lengths.

Given two histograms H_1 and H_2, a similarity score is computed as follows:

$$S(H_1, H_2) = c - \sum_{i=1}^{L} \frac{[H_1(i) - H_2(i)]^2}{H_1(i) + H_2(i)}$$

where L is the number of bins and c is a constant (actually, it is the average similarity across all training traces). Note that summation is over all bins where either of the histograms has non-zero value. Given a similarity score between any two traces, a similarity matrix is constructed across all the representative traces.

For a similarity measure to be a kernel function, the similarity matrix should be positive semidefinite (psd).

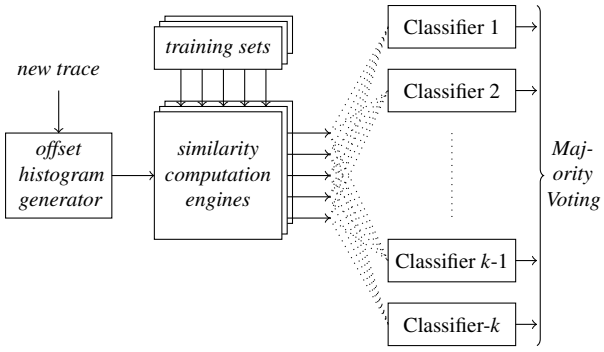

Figure 1: Block diagram for m types of workload classification. Number of classifiers $k = \frac{1}{2}m(m-1)$.

In our application we construct a psd kernel matrix by setting the negative eigen-values of the similarity matrix to zero[3].

3.2 Binary Classification

Consider a binary classification problem where only 2 kinds of application workloads, with labels $+1$ and -1, generate the traces. With each trace $\mathbf{x}$ we associate a label $\mathbf{y} \in \{+1,-1\}$ to represent its class. Now the workload identification problem can be formally stated as: given a trace $\mathbf{x}$, infer the value of $\mathbf{y}$.

Assume a *training set* consisting of representative traces from each class is given. Let $\mathbf{x}_1 \ldots \mathbf{x}_n$ be the training traces with known class labels $\mathbf{y}_1 \ldots \mathbf{y}_n$. Using this training set we compute the Kernel, $K(\cdot,\cdot)$, as explained in Section 3.1.

Given a new trace $\mathbf{x}$, the SVM classifier function f produces a classification score as:

$$f(\mathbf{x}) = b + \sum_{i=1}^{n} \alpha_i \mathbf{y}_i K(\mathbf{x}_i, \mathbf{x}),$$

where the coefficients $\alpha_1, \ldots, \alpha_n$ and the bias term b are found by an SVM training algorithm. Now, the decision value $\hat{\mathbf{y}}$ is calculated as:

$$\hat{\mathbf{y}} = \begin{cases} +1 & \text{if} \quad f(\mathbf{x}) > 0, \\ -1 & \text{otherwise}, \end{cases} \tag{1}$$

3.3 Multi-class Classification

Consider multi-class classification as shown in Figure 1. The *similarity computation engine* computes the Kernel matrix using the *training sets* for m workload classes. With $m > 2$, the binary classification scheme can be extended by creating one classifier for every pair of classes for a total of $\frac{1}{2}m(m-1)$ classifiers. Given a set of representative traces from each class, a classifier can be found for every pair of classes by treating the representative traces from one class as positive type and the representative traces from the other class as negative type. Let $f_{(i,j)}$, $1 \le i < j \le m$, be the decision function when class-i is considered as positive class and class-j as negative. Let t be a chosen non-negative number. In order to predict the workload class for a trace $\mathbf{x}$, compute all the decision values $\{ f_{(i,j)}(\mathbf{x}) \mid 1 \le i < j \le m \}$ and corresponding vote $\hat{\mathbf{y}}_{(i,j)}$ as

$$\hat{\mathbf{y}}_{(i,j)} = \begin{cases} i & \text{if} \quad f_{(i,j)}(\mathbf{x}) > t, \\ j & \text{if} \quad f_{(i,j)}(\mathbf{x}) \le -t, \end{cases} \tag{2}$$

Based on all the $\hat{\mathbf{y}}_{(i,j)}$ values we calculate the number of votes in favor of each class and we classify the trace $\mathbf{x}$ to the class which gets maximum number of votes. Here we set threshold $t = 0$.

We would also like to identify traces that do not belong to the m trained classes. To do so, we choose a suitable threshold $t > 0$ and classify a trace $\mathbf{x}$ to class $i \in \{1,2,\ldots,m\}$ if and only if number of votes in its favor is exactly $m-1$; otherwise we declare its class as *unknown*.

3.4 Online Self-correction

To improve LoadIQ's ability to classify untrained workloads as unknown, in an online deployment, over time, we collect trace snippets that the SVM based multi-class classifier flags as 'unknown'. We label them with a special "unknown" class label and re-train LoadIQ augmented with this class and re-classify past snippets to see if any of them join this class. As Section 4.1 shows, this works well in practice as it exploits LoadIQ's ability to distinguish workloads with explicitly identified features.

4 Evaluation

Several enterprise-class applications have distinct phases of behavior that require specific storage optimizations not applicable at all times. However, as these phases may occur aperiodically, it is cumbersome to manually schedule their storage-level optimizations. In this section, we test LoadIQ to detect such application phase boundaries to enable phase specific storage optimizations. First, we use LoadIQ to identify the distinct phases of TPC-DS benchmark on PostgreSQL database. We show that LoadIQ is able to distinguish between various phases with high accuracy. We also demonstrate the self correction ability of LoadIQ using *unknown* class. We then use LoadIQ to perform online labeling of a production OLAP workload. This experiment demonstrates how recurrence of special/anomalous workload behavior can be spotted by administrators using LoadIQ.

4.1 Distinguishing Phases in a Large-scale Database Workload

A typical database workload consists of four phases: The *load* phase populates the database tables. We use the PostgreSQL COPY commands for this. The *indexing*

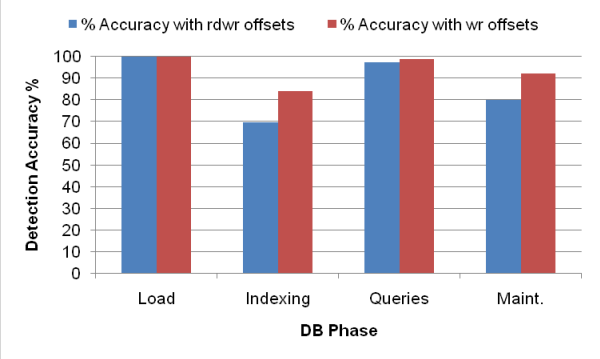

Figure 2: Accuracy of identifying DB Workload phases when fully trained using all offsets in a trace vs. only write offsets. Write offsets yield consistently better accuracy, so we switch to them for subsequent experiments.

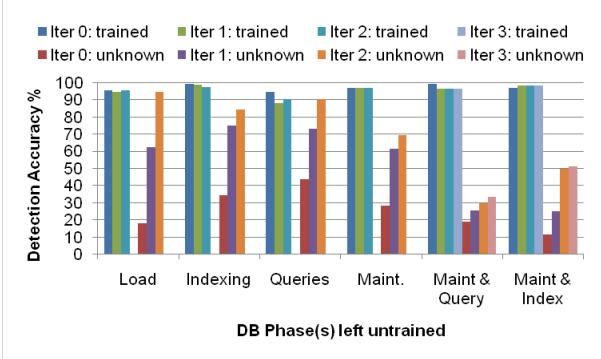

Figure 3: Accuracy of identifying DB Workload phases from write offsets when partially trained. Re-training iteratively improves accuracy for untrained phases.

phase creates various desired indexes on the tables after *load* and also in response to the administrator requests. In the *query* phase, the database serves client queries. We choose from 99 different TPC-DS query templates to issue these queries. Finally, the *maintenance* phase gets triggered by a database engine periodically or on demand for clean up.

In our experimental setup, the database runs inside a guest VM with 4GB RAM whose image resides on an NFS server. The VM's host machine is an 8-core Xeon-5520 with 8GB RAM. We use the TPC-DS dataset generator to populate the database with a 2GB dataset, which results in a 3.5x bloat in size on disk.

Training: For training LoadIQ, we collect traces while the database goes through various phases and label each trace with its phase name, manually. Just for training, we collect load phase traces when generating TPC-DS datasets of size 1, 2, 3 and 5GB. During indexing phase, we create three types of indexes: B-tree, GIN, Hash. For maintenance phase, we run PostgreSQL's VACUUM and ANALYZE commands. To generate a query phase trace, we run a random mix of queries selected from all the 99 TPC-DS query templates in 100 concurrent connections. We divide the traces collected into 60-second snippets, generate read-write histograms for each to be used in building the SVM model.

Results: Figure 2 shows the classification accuracy when LoadIQ is trained to identify all classes. The accuracy is better ($> 84\%$) when write offset shift histogram was used instead of combined offset histogram (70%) as the discriminating feature for classification. LoadIQ also flagged the remaining traces as "unknown" instead of confusing them as another known class.

Figure 3 shows how classification accuracy for unknown workloads starts off very low, but improves rapidly with re-training. When LoadIQ was trained to detect three out of four phases and tested with the fourth phase, it correctly labeled only about 17-43% of the

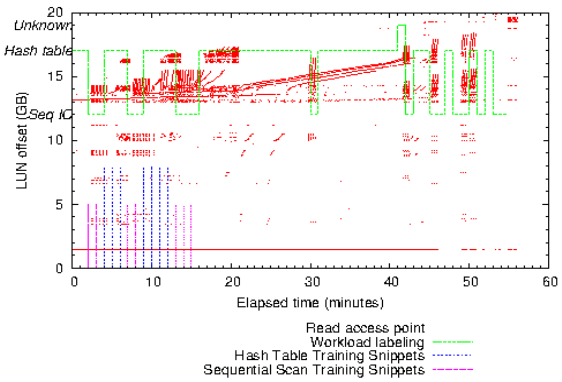

Figure 4: Detecting User-identified phases in an Enterprise-class OLAP application: LoadIQ can be trained to spot hash table accesses and streaming IO bursts.

traces as unknown, and misclassified the rest of them. Thus LoadIQ's core SVM engine is able to only distinguish among known classes. However, with just two rounds of re-training as explained in Section 3.4, LoadIQ is able to flag more than 80% of untrained single workloads correctly as unknown. The improvement is not as dramatic when LoadIQ encounters a mix of multiple untrained workloads.

4.2 Live Labeling of a Production OLAP Workload

In this section, we demonstrate how an administrator can use LoadIQ to automate detecting the recurrence of special/anomalous workload behaviors in a production environment simply by pointing out a few time windows when they occurred in the past. For this experiment, we run a production enterprise data warehousing application in a 10-node cluster configured to use a SAN backend. The application's data is spread over 50 LUNs each of size 20GB. We capture the post-host-cache SCSI request trace on all LUNs - 188K reads and 250K writes per LUN spread over 56 minutes. Figure 4 shows a scatter plot of the reads issued on one such LUN.

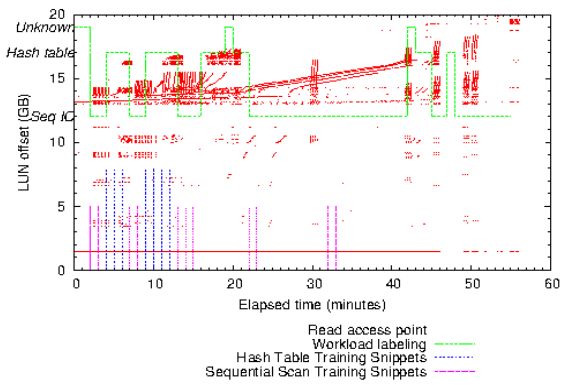

Figure 5: Enterprise-class OLAP application: With a little extra user annotation, LoadIQ spots slow interleaved sequential IO as well.

The admin identifies three lumps of points in the 16-18GB region during the 6-20 min. interval, and recognizes them as accesses to a growing hash table. He also sees lumps of near-vertical lines (e.g., 2-4 and 8-10 mins.) and identifies them as sequential IO bursts.

We use LoadIQ to identify hash table accesses (labeled *Hash table*) and sequential IO bursts (labeled *Seq IO*) from the trace, after training it with a few initial 1-minute trace snippets for each class shown in Figure 4 as impulses. The figure shows how LoadIQ labeled the trace as a square wave with labels on the left in italics. It identifies all the hash table accesses and the fast sequential IO bursts accurately, except during the 40th and 50th minutes. It classified the concurrent sequential IOs (slanted lines) during 20-40 minute period as random IO, because their offset shifts look similar to random IO.

Next, the admin labels four extra snippets in the 20-40 min. interval as sequential IO and re-trains LoadIQ. Figure 5 shows that LoadIQ identifies sequential IO as such correctly while continuing to identify hash table accesses.

Live Labeling To evaluate whether the same labeling can be done online at real-time, we feed the live SCSI trace of the above workload to LoadIQ running on a separate host (8-core Xeon-5520 with 8GB RAM), in 60-second chunks. LoadIQ emits the same labels as before, every minute within 4 seconds. The retraining step takes about 4 seconds. Thus, LoadIQ can detect a workload shift within minutes.

5 Discussion and Conclusions

In this paper, we present LoadIQ, a machine-learning-based tool for identifying various phases in an application, from its live storage trace, at accuracy $> 93\%$ in many cases. Also, if LoadIQ classifies a workload as 'unknown', it is very unlikely to be a known workload. However, LoadIQ's accuracy at flagging untrained workloads as unknown is poor without retraining. This is because it has no reference to know what an 'unknown' workload looks like. LoadIQ's iterative self-correction alleviates this problem by learning to identify untrained workloads, as long as their patterns do not shift too rapidly.

For LoadIQ to be truly dependable in real deployments, it needs a quantifiable confidence measure of its classification output. That is an area of future work. Also, further study is needed to assess how variable concurrency and caching at client-level affects LoadIQ's ability to identify various application phases.

References

[1] E. Anderson. Capture, conversion and analysis of an intense nfs workload. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST 2009)*, Feb. 2009.

[2] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. Borg: Block-reorganization for self-optimizing storage systems. In M. I. Seltzer and R. Wheeler, editors, *FAST*, pages 183–196. USENIX, 2009.

[3] Y. Chen, M. R. Gupta, and B. Recht. Learning kernels from indefinite similarities. In *International Conference on Machine Learning*. 2009.

[4] D. Ellard. *Trace-based analyses and optimizations for network storage servers*. PhD thesis, Cambridge, MA, USA, 2004. Adviser-Margo I. Seltzer.

[5] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST03)*, pages 203–216, 2003.

[6] D. Gu and C. Verbrugge. A survey of phase analysis: Techniques, evaluation and applications. Technical report, Citeseer, 2006.

[7] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and analysis of large-scale file system workloads. In *Proceedings of the USENIX 2008 Annual Technical Conference*, June 2008.

[8] T. Madhyastha and D. Reed. Input/output access pattern classification using hidden markov models. In *Workshop on Input/Output in Parallel and Distributed Systems*, Nov. 1997.

[9] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX 2006 Annual Technical Conference*, June 2006.

[10] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond (Adaptive Computation and Machine Learning)*. The MIT Press, 2001.

[11] N. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla. Discovery of application workloads from network file traces. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST 2010)*, Feb. 2010.

[12] J. Zhang, M. Yousif, R. Carpenter, and R. Figueiredo. Application resource demand phase analysis and prediction in support of dynamic resource provisioning. In *Fourth International Conference on Autonomic Computing, 2007.*, pages 12–12, 2007.

[13] X. Zhang, A. Riska, and E. Riedel. Characterization of the e-commerce storage subsystem workload. In *QEST*, pages 297–306. IEEE Computer Society, 2008.