Characterizing Tenant Behavior for Placement and Crisis Mitigation in Multitenant DBMSs

Aaron J. Elmore[‡] Sudipto Das[$ *] Alexander Pucher[‡]
Divyakant Agrawal[‡] Amr El Abbadi[‡] Xifeng Yan[‡]

[‡]University of California, Santa Barbara
Santa Barbara, CA, USA
{aelmore,pucher,agrawal,amr,xyan}@cs.ucsb.edu

[$]Microsoft Research
Redmond,WA USA
sudipto.das@microsoft.com

ABSTRACT

A multitenant database management system (DBMS) in the cloud must continuously monitor the trade-off between efficient resource sharing among multiple application databases (tenants) and their performance. Considering the scale of hundreds to thousands of tenants in such multitenant DBMSs, manual approaches for continuous monitoring are not tenable. A self-managing controller of a multitenant DBMS faces several challenges. For instance, how to characterize a tenant given its variety of workloads, how to reduce the impact of tenant colocation, and how to detect and mitigate a performance crisis where one or more tenants' desired service level objective (SLO) is not achieved.

We present **Delphi**, a self-managing system controller for a multitenant DBMS, and **Pythia**, a technique to learn behavior through observation and supervision using DBMS-agnostic database level performance measures. Pythia accurately learns tenant behavior even when multiple tenants share a database process, learns good and bad tenant consolidation plans (or packings), and maintains a per-tenant history to detect behavior changes. Delphi detects performance crises, and leverages Pythia to suggests remedial actions using a hill-climbing search algorithm to identify a new tenant placement strategy to mitigate violating SLOs. Our evaluation using a variety of tenant types and workloads shows that Pythia can learn a tenant's behavior with more than 92% accuracy and learn the quality of packings with more than 86% accuracy. During a performance crisis, Delphi is able to reduce 99th percentile latencies by 80%, and can consolidate 45% more tenants than a greedy baseline, which balances tenant load without modeling tenant behavior.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases*

Keywords

Multitenancy; elastic data management; database consolidation; tenant characterization; shared nothing architectures

*Work done while at UCSB

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

1. INTRODUCTION

Cloud application platforms and large organizations face the challenge of managing, storing, and serving data for large numbers of applications with small data footprints. For instance, several cloud platforms such as Salesforce.com, Facebook, Google AppEngine, and Windows Azure host hundreds of thousands of small applications. Large organizations, such as enterprises or universities, also face a similar challenge of managing hundreds to thousands of database instances for different departments and projects. Allocating dedicated and isolated resources to each application's database is wasteful in terms of resources and is not cost-effective. **Multitenancy** in the database tier, i.e., sharing resources among the different applications' databases (or **tenants**), is therefore critical. We focus on such a multitenant database management system (**DBMS**) using the **shared process multitenancy** model where the DBMS comprises a cluster of database servers (or **nodes**) where each node runs a single database process which multiple tenants share.

1.1 Challenges in Multitenancy

A multitenant DBMS must minimize the impact of colocating multiple tenants. The challenge lies in determining *which* tenants to colocate and *how many* to colocate at a given server, i.e., learn good **tenant packings** that balance between over-provisioning and over-booking. Furthermore, the colocated tenants' resource requirements must be complementary to avoid heavy resource contention after colocation.

To ensure of service, a multitenant DBMS must also associate meaningful **service level objectives** (**SLOs**) in a consolidated setting. If a tenant's SLO is violated, the DBMS must adapt to this **performance crisis**. The challenge lies in mitigating the crisis which might be caused by a change in this tenant's behavior, a change in a colocated tenant's behavior, or a degradation in the node's performance. A tenant's behavioral change might be due to a change in the query pattern, data access distribution, working set size, access rates, or queries issued on non-indexed attributes while typical queries are on indexed attributes—complexity arises from the myriad of possibilities. Adapting to a crisis entails detecting changes, filtering erratic behavior, and devising mitigation strategies. Erratic behavior can arise from temporary shifts in application popularity, periodic analysis, or ad-hoc queries.

The problem of designing a self-managing controller is further complicated by the variety of tenant workload types. Many applications use their databases for multiple purposes, such as using the same database for serving, analysis, and logging. Therefore, in addition to workload variations across tenants, a single tenant might also exhibit different behaviors at different time instances. Behavioral changes might have patterns (e.g., diurnal trends of serving

and reporting workloads) or might be erratic (e.g., flash crowds). Moreover, dynamics in the workload might or might not have correlation. For instance, hosted business applications observe a spike in multiple tenants' activity at the start of the business day. These behavioral dynamics, the interplay of shared resources among colocated tenants, and the complex interactions between the DBMS, OS, and the hardware make analytical models and theoretical abstractions impractical.

From the monitoring perspective, a system controller potentially receives hundreds of raw performance measures from the database process and the operating system (OS) at each node. Considering the scale of tens to hundreds of nodes, using all these raw signals to maintain an aggregate view of the entire system and the individual tenants results in an information deluge. One challenge in effective administration is to systematically filter, aggregate, and transform these raw signals to a manageable set of attributes and automate administration with minimal human guidance. An intelligent and self-managing system controller is a significant step towards achieving economies-of-scale and simplifying administration.

More than a decade of research has focused on effective multitenancy at different layers of the stack, including sharing the file system or the storage layer [15, 19], sharing hardware through virtualization [10], and in the application and web server layer [21]. Multitenancy in the database tier introduces novel challenges due to the richer functionality supported by the DBMS compared to the storage layer, and the complex interplay between CPU, memory, and disk I/O bandwidth observed in a DBMS compared to the stateless applications and web servers. Recent work has focused on various aspects of database multitenancy. Kairos [6] is a technique for tenant placement and consolidation for a set of tenants with known static workloads. Kairos uses direct measurements of the tenants' CPU, I/O, memory, and disk resource consumption to suggest consolidation plans. SmartSLA [24] is a technique for cost-aware resource management using direct resource utilization measurements to learn the average SLA penalty cost in a setting where each tenant has its independent database process and virtual machine. Ahmad and Bowman [1] use machine learning techniques to predict aggregate resource consumption for various workload combinations and proposes a technique that relies on static and known workloads. The authors argue that analytical models for performance and consolidation are hard due to complex component interactions and shifting bottlenecks. Lang et al. [14] propose a SLO-focused framework for static provisioning and placement where tenant workloads are known. In general, existing approaches do not target the problem of continuous tenant modeling, dynamic tenant placement, variable and unknown tenant workloads, and performance crisis mitigation in the shared process multitenancy model, which is critical for deploying shared database services.

1.2 Controller for a Multitenant DBMS

We present the design and implementation of **Delphi**, an intelligent self-managing controller for a multitenant DBMS that orchestrates resources among the tenants. Delphi uses **Pythia**, a technique to *learn* behavior through observation.[1] Pythia uses DBMS-agnostic database-level performance measures available in any standard DBMS and supervised learning techniques to learn a **tenant model** representing resource consumption. Pythia learns a **node model** to determine which combination of tenant types perform well after colocation (**good packings**) and which combinations do not perform well (**bad packings**). Pythia continuously models behavior and maintains historical behavior which allows it to detect a change in a tenant's behavior. Once Delphi detects a performance crisis, it leverages Pythia to suggest remedial actions. Identifying a set of tenants to relocate, and finding destinations for these tenants to alleviate latency violations is the core challenge addressed by Pythia. Delphi employs a local search algorithm, hill-climbing, to prune the space of possible tenant packings and uses the node model to identify potential good packings. Pythia requires minimal human supervision, typically from a database administrator, only for training the supervised learning. Once the models are trained, Delphi can independently orchestrate the tenants, i.e., monitor the system to detect performance crises, load-balance and migrate tenants to mitigate a crisis and to ensure that tenant SLOs are being met. Figure 1 presents an overview of Delphi's design.

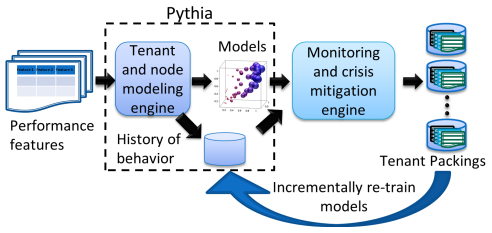


Figure 1: Pythia incrementally learns behavior.

In contrast to existing techniques that directly use OS or VM level resource utilization, such as Kairos [6] and SmartSLA [24], Pythia uses database-level performance measures such as cache hit ratio, cache size, read/write ratio, and throughput. This allows Pythia to maintain a detailed per-tenant profile even when tenants share a database process. OS level measures either provide aggregate resource consumption metrics of all tenants, and the alternative of hosting one tenant per database process degrades performance [6]. In contrast, Pythia results in negligible performance impact by using performance measures available from any standard DBMS implementation. Additionally, Pythia learns tenant behavior without any assumptions or in-depth understanding of the underlying systems. In addition, unlike workload driven techniques [1,6,14], Pythia does not require advanced knowledge of the tenants' workload or limit the workload types. Moreover, Pythia does not require profiling tenants in a sandbox, a dedicated node for running tenants in isolation, thus making it applicable even in scenarios where production workloads cannot be replayed due to operational or privacy considerations [2]. Therefore, we expect Pythia to have applications in a variety of multitenant systems and environments, while requiring minimal changes to existing systems.

Delphi is the first end-to-end framework for the accurate and continuous modeling of tenant behavior in a shared process multitenancy environment. We built a prototype implementation of Delphi in a multitenant DBMS running a cluster of Postgres RDBMS servers. Our current implementation uses a set of **classifiers** to learn tenant and node models, although Pythia can be extended to use additional tenant resource models, or other machine learning techniques such as clustering or regression learning [23]. Pythia learns tenant models with a 92% accuracy, and node models with a 86% accuracy. Once a performance crisis is detected, Delphi can mitigate the crisis by reducing the 99th percentile latency violations by 80% on average.

This paper makes the following major contributions:

- Pythia, a technique to accurately learn tenant behavior using database-level attributes agnostic of a specific DBMS implementation. Pythia dynamically detects changes to a tenant's behavior even when tenants share a database process, and automatically learns good and bad tenant packings through observation.

[1]Delphi is an ancient Greek site, where the oracle Pythia resided.

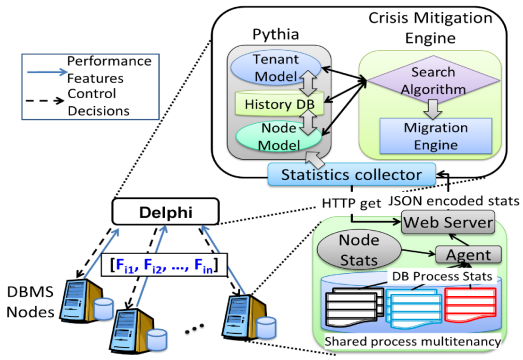


Figure 2: Overview of Delphi's architecture.

- An end-to-end design and implementation of Delphi, an intelligent self-managing controller for a multitenant DBMS. Delphi non-intrusively collects system wide statistics, load-balances to mitigate performance crises, and ensures tenant SLOs.
- A detailed evaluation and comparison against a greedy baseline, using a custom distributed multitenant benchmarking framework.

Organization: Section 2 provides background on database multitenancy and formulates the problem. Section 3 explains how Pythia learns the tenant and node models. Section 4 presents the detailed end-to-end implementation of Delphi. Section 5 presents a thorough evaluation using a variety of workloads and tenant types. Section 6 surveys the related work and Section 7 concludes the paper.

2. DATABASE MULTITENANCY

2.1 Multitenancy Models

Different multitenancy models arise from resource sharing at different levels of abstraction; the **shared hardware**, **shared process**, and **shared table** models are well known [13]. Salesforce.com [22] uses the shared table model where tenants share the database tables. SQL Azure [3] and Relational Cloud [7] use the shared process model where tenants share the database process. Xiong et al. [24] use the shared hardware model where tenants share the physical hardware but have independent VMs and database processes.

The choice of multitenancy model has implications on a system's performance, consolidation, and dynamics. For instance, the shared hardware model provides strong VM level isolation and elastic scaling. However, as shown in a recent study [6], such a model results in up to an order of magnitude lower performance and consolidation compared to the shared process model. On the other hand, the shared table model allows efficient resource sharing amongst tenants but restricts the tenant's schema and requires custom techniques for efficient query processing. The shared process model, therefore, provides a good balance of effective resource sharing, schema diversity, performance, and scale. The shared process model has also been widely adopted in commercial and research systems [3, 7]. We therefore focus on the design of an autonomic controller for the shared process multitenancy model.

2.2 Delphi Architecture

Figure 2 provides an overview of the DBMS architecture we consider in this paper. The system consists of a cluster of shared-nothing DBMS nodes where each node runs an RDBMS engine which multiple tenants share. Delphi is the overall framework for a self-managing system controller and comprises Pythia, which models tenant and node behavior and maintains historical information, a **crisis detection and mitigation engine**, and a **statistics collector**, which gathers system-wide performance statistics.

Every DBMS node has a lightweight **agent** which collects usage statistics at that node. A lightweight embedded web server interfaces the agent to the statistics collector. Delphi periodically requests a snapshot of the performance measures from all nodes. On receipt of a snapshot request, the agent obtains per-tenant and aggregate performance statistics from the database process and the OS, collectively called the **performance features**. The agent then returns the snapshot to Delphi. We choose a pull based approach over a push based approach where the agents periodically push the snapshot to the Delphi. A pull based approach enables an agent to operate without any persistent state information and allows the snapshot frequency to be configured by Delphi. The agent collects and aggregates statistics only when requested by Delphi.

All snapshots collected by Delphi are stored in a **history database** that stores the per-tenant and per node history and serves as a persistent repository for historical information such as tenant behavior, packings, and actions taken by Delphi to mitigate a performance crisis. The history database can also be used for off-line analysis of Delphi's actions and performance by an administrator.

2.3 Service Level Objectives

In order for a shared multitenant DBMS to be attractive for the tenants, the DBMS must support some form of performance guarantees such as response times. Typically, the tenants and the DBMS provider agree on a set of service level agreements (**SLAs**). However, agreeing on SLAs is typically an arms race between the provider and the tenant often governed by business logic [2]. Automatically assigning performance SLAs based on workload characteristics is a hard problem and beyond the scope of this paper. Instead, we focus on service level objectives (SLOs) as a mechanism for quantifying the quality-of-service of a multitenant DBMS for a given tenant's workload. We rely on using a uniform percentile-based latency SLOs for all tenants.

2.4 Effects of Colocation

In the shared process multitenancy model, tenants share the DBMS, OS, and hardware. This includes sharing the DBMS's **buffer pool**, the OS's file system cache (or **page cache**), the available I/O bandwidth, and CPU cycles. DBMSs are typically optimized for scaling-up a single tenant and are not designed to fairly share resources among tenants. Therefore, it is critical to understand the impact of resource sharing and contention on performance. It is well understood that when colocating multiple tenants, no resource should be over-utilized. Approaches, such as Kairos [6], are known to determine which tenants can be colocated based on the tenant's resource consumption. Furthermore, it is also imperative that colocated tenants have complementary resource consumption. As an example, colocating a mix of disk-heavy and CPU-heavy tenants is probably better than colocating multiple disk-heavy tenants.

In addition to the above heuristics, when multiple tenants share the cache, it is important to consider how the tenants access the cache. Assume we have two tenant databases D_1, and D_2 colocated at a node with enough resources to serve both tenants. If D_1's throughput is higher than that of D_2, the least recently used (**LRU**) eviction policy, commonly used in buffer pool implementations, will result in D_1 stealing cache pages from D_2. This is because D_1 is accessing more pages, thus making D_2's pages candidates for likely eviction. This reduction in D_2's cache hit ratio will affect its performance. Figure 3 demonstrates this behavior in

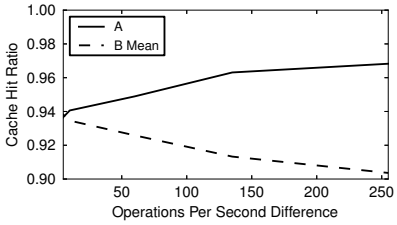


Figure 3: Effects of throughput on cache impedance.

an experiment on a node serving a set of identical tenants where a tenant A has its throughput gradually increasing while the other tenants (B) have a fixed throughput; B represents the average behavior of the remaining tenants. As the throughput difference increases, A steals more cache from the other tenants resulting in an increase in A's cache hit ratio and a decrease in the cache hit ratio of all the remaining tenants (B-Avg), thus affecting performance of the slower tenants.

We introduce the concept of **cache impedance** as a measure of compatibility between tenants to effectively share a cache. A scenario where one tenant dominates the cache over the other tenants is due to a cache impedance mismatch. Therefore, to ensure that tenants continue to perform well after colocation, it is important their cache impedance matches. Four major aspects affect cache access: key or tuple access distribution, database size, throughput, and the read/write distribution; though from our experiments, we have seen throughput to be the most dominant factor.

A similar impact of cache impedance is also observed for accesses to the OS page cache and how the tenants share the page cache. However, the interactions between the two levels of caches, the buffer pool and page cache, are complex due to the stateful interactions between the OS and DBMS [1]. We therefore take an empirical approach to infer a tenant's cache impedance through observed behavior, thus avoiding solutions tailored to a specific implementation. We incorporate the knowledge of cache impedance into the design of Pythia by using signals from both the buffer pool and the page cache to determine the class labels that represent tenant behavior. Pythia learns which combinations of tenant classes are amenable for colocation. While specialized solutions can be built with strong resource isolation between tenants or bypassing the page cache using direct I/O, we aim to present an approach requiring minimal changes to current DBMS architectures.

2.5 Problem Formulation

Let $\mathbb{D}$ be the set of tenant databases and $\mathbb{N}$ be the set of DBMS nodes. Each tenant $D_i \in \mathbb{D}$ is represented by a vector of performance features F_i and $\mathbb{F} = \{F_i | D_i \in \mathbb{D}\}$. Let $\mathcal{C}$ be a set of class labels corresponding to tenant behavior. Pythia learns the tenant model $\mathbb{T} : \mathbb{F} \rightarrow \mathcal{C}$, i.e., given a performance feature F_i of tenant D_i, the tenant model $\mathbb{T}$ assigns a label $c \in \mathcal{C}$ to D_i. Let P_j be the set of tenants at N_j, such that each tenant entry, D_i, has a corresponding class label $\mathcal{C}_i$ derived from $\mathbb{T}$ and performance measure (e.g.,latency), M_i, that constitutes the SLO. Pythia also learns the node model $\mathbb{P} : P_j \rightarrow \mathcal{G}$, i.e., given a set of tenants P_j, and thus corresponding labels, $\mathbb{P}$ assigns a label $g \in \mathcal{G}$ that indicates the packing's quality. We set $\mathcal{G} = \{\checkmark^+, \checkmark, \times\}$, where $\checkmark$ denotes a good packing, $\checkmark^+$ is a good packing with under-utilized resources, and $\times$ is a bad packing.

Delphi periodically collects snapshots from all the DBMS nodes. Each node reports the performance features of all the tenants hosted at that node and the overall node level usage statistics. Delphi monitors all the nodes and ensures that all the tenants' SLOs are being met and every node in the system is performing well, i.e., $\forall N_j \in \mathbb{N}, \mathbb{P}(P_j) \equiv \checkmark$. For every tenant, Pythia maintains a sliding window of the last $\mathbb{W}$ snapshots and their corresponding labels. Given a performance crisis at a node N_j, i.e., one or more tenants in the packing P_j are violating their SLOs, Delphi mitigates the crisis by finding a packing $P'_j \subset P_j$ such that $\mathbb{P}(P'_j) \equiv \checkmark$. Delphi also finds destination node(s) (N_k) for the tenants $P_j - P'_j$ that must be moved out of N_j.

3. PYTHIA: LEARNING BEHAVIOR

Pythia seeks to learn tenant and node models through observation. Using machine learning classification, Pythia assigns a class label to a tenant that approximately describes the tenant's behavior and resource consumption. Pythia also learns which tenant packings are performing well and which ones are violating SLOs. Pythia's design goals are to learn behavior while: (i) requiring minimal changes to the underlying multitenant DBMS, (ii) having no foreknowledge or assumptions (such as static or predefined workloads or working set fitting in memory) on tenant workloads, and (iii) causing negligible performance impact.

Pythia uses supervised learning techniques, specifically classifiers, to learn tenant behavior. We use tenant classes that are representative of their resource consumption. Pythia also learns which combination of tenant types perform well together (good packings) and which tenant packings are likely to over consume resources or violate latency SLOs (bad packings). We now explain our feature selection process to identify a small set of performance features that can accurately model a tenant's behavior and then explain how Pythia uses these features to learn the tenant and node models.

3.1 Tenant Feature Selection

Pythia uses DBMS-agnostic database-level performance measures. Database-level performance measures allow per-tenant monitoring for detailed analysis even in shared process multitenancy. A plethora of performance measures can be extracted from any standard DBMS. Examples are number of cache accesses and cache misses, number of read/update/insert/delete requests, number of pages accessed by each transaction, average response time, and transactions per second. However, using all the measures to characterize a tenant is not desirable since the complex relationship between attributes can be difficult for classifiers to infer. We therefore use our domain knowledge to select a subset of measures.

The challenge lies in selecting the measures that correlate to tenants' behavior and resource requirements while ensuring high modeling accuracy. To allow Pythia to be used in a large variety of systems, we only select measures available from any standard DBMS implementation, be it SQL or NoSQL, with negligible or no impact on normal operation. We select the feature set guided by our knowledge of the attributes' semantics.

We now explain the different measures, some of which are derived from other raw measures, we choose as the tenant performance features and also explain the rationale behind their selection.

Write percent. The percentage of operations issued by a tenant that are writes, i.e., inserts, deletes, and updates. This measure gives an estimate of the rate at which the pages are updated and is also an indirect indicator of the expected disk traffic resulting from the writes due to cache pages being flushed, checkpointing, and appends to the transaction log.

Average operation complexity. Average number of (distinct) database pages accessed by a single transaction. When transactions are specified in a declarative language, such as SQL, operation complexity is a measure of the resources consumed (such as CPU cycles) to execute that operation. This measure differentiates a tenant issuing simple read/write transactions from one issuing complex

transactions performing joins or scans, even though both transaction types might issue the same number of SQL statements.

Percent cache hits. Number of database pages accessed that were served from the cache. This measure approximates the number of disk access requests issued by the database process for the tenant.

Buffer pool size. Number of pages allocated to the tenant in the DBMS buffer pool, approximates the tenant's memory footprint.

OS page cache size. Number of pages allocated in the OS page cache to the tenant's database files. Along with buffer pool size, provides an estimate of the tenant's total memory footprint.

Database size. Size of the tenant's persistent data and is representative of the disk storage consumption. Size has an indirect impact on the tenant's disk I/O.

Throughput (Transactions per second). Average number of transactions completed for a given tenant in a second. Transactions completed include both the transactions committed and rolled-back. Throughput is an indicator of behavior (such as cache impedance) and resource requirements (such as CPU and disk bandwidth).

Our experiments (Section 5) demonstrate that these attributes allow Pythia to train accurate models, representative of resource consumption, that can be used by Delphi for effective crisis mitigation.

3.2 Resource-based Tenant Model

We characterize a tenants' behavior in terms of its expected resource consumption. The rationale for selecting resource-based models is to allow Pythia to associate semantics to a tenant's behavior and reason about resources consumed. Furthermore, the number of critical resources are limited and known to a system administrator. Therefore, tenant behavior can be classified into a handful of resource-based classes without requiring knowledge of tenant workloads. Recent work has investigated classifying workload-based modeling by running queries in isolation [18]. However, using queries to build accurate models and understand tenant interaction, assumes new queries follow existing patterns and limits the opportunity for ad-hoc queries. Additionally, queries that rely on caching to minimize expensive operations, such as nested loop joins, can make modeling interactions difficult. In designing Pythia we sought to avoid these assumptions. We now explain how we select the resource-based class labels, how we determine the rules to assign labels to the tenants, and how we train the tenant model.

3.2.1 *Resource-based classes*

A tenant's resource consumption has four important components: CPU consumed to fulfill the tenant's requests, main memory (RAM) consumed to cache data, disk capacity consumed to store persistent data, and the disk I/O bandwidth (or disk **IOPS**) consumed. While networking is an important resource, for now we assume that database connections are ample due to connection pooling and workloads use minimal data transfer. In scenarios where this is critical, network attributes can be be added. In most cases, commodity servers have disks (order of terabytes) much larger than the tenants (order of few gigabytes). Therefore, disk capacity is almost always abundant. However, irrespective of how much RAM is provided, the DBMS and the OS in a running server will invariably use most of the available RAM to cache existing tenants. Therefore, RAM usage is almost always constrained and excess capacity for new tenants is not reserved. However, in practice, a large fraction of the cached pages are not actively used and a newly added tenant will carve out cache capacity, provided the tenant's cache impedance, as described in Section 2.4, matches that of the tenants already being served at the node. In addition to cache impedance, the disk IOPS and the available CPU capacity are two critical resources that determine how well the tenants perform after colocation. In a consolidated setting, we cannot directly determine the exact CPU and disk IOPS consumption or the cache impedance of each tenant, so we model them indirectly. We use the performance features to approximate resource consumption. We base our class labels based on three dimensions: expected disk IOPS (`D`), throughput (`T`), and operation complexity (`O`). We loosely associate `D`, `T`, and `O` to disk bandwidth consumption, cache impedance, and CPU consumption.

A human administrator is required to derive resource boundaries for the labels. These boundaries can be derived by observing resource distributions in a system where Pythia will be used. In our evaluation, we partition each continuous dimension into a few buckets whose boundaries are determined by analyzing the distribution of values obtained from an operational system. The `D` dimension is subdivided into four buckets: small (`DS`), medium (`DM`), large (`DL`), and extra-large (`DXL`). The `T` dimension is subdivided into three buckets: small (`TS`), medium (`TM`), and large (`TL`). The `O` dimension is subdivided into two buckets: small (`OS`) and large (`OL`). The rationale for such a subdivision is that the disk bandwidth is a critical resource for data intensive applications; a finer subdivision allows closer monitoring of a tenant's disk bandwidth consumption. Throughput impacts CPU consumption, disk IOPS, and cache impedance. We therefore consider throughput second after disk and use a coarser subdivision into three buckets. Operation complexity affects CPU consumption and is primarily targeted to differentiate tenants issuing complex queries such as scans, reports, or joins. Complexity comes last and is subdivided into two buckets.

We use class labels composed of `D`, `T`, and `O`. For instance, `DS-TS-OS` represents a tenant with low expected disk consumption, low throughput, and low complexity. Among the possible 24 classes, in practice, some classes fold into one encompassing class as some dimension override the others. As an example, a `DXL` tenant's resource consumption is high enough that the operation complexity does not matter; we only consider two throughput buckets for such tenants: `DXL-TL` and `DXL-TMS` (medium and small). Similarly, for a `DL-TL` tenant, complexity is irrelevant. Following is the set of class labels we used: $\mathcal{C}$ = {`DL-TL, DL-TM-OL, DL-TM-OS, DL-TS-OL, DL-TS-OS, DM-TL-OL, DM-TL-OS, DM-TM-OL, DM-TM-OS, DM-TS-OL, DM-TS-OS, DS-TL-OL, DS-TL-OS, DS-TM-OL, DS-TM-OS, DS-TS-OL, DS-TS-OS, DXL-TL, DXL-TMS`}.

3.2.2 *Training the model*

Throughput and operation complexity are directly measured from the database process. The number of disk requests issued per-tenant must be estimated since the OS only provides aggregate measures and the DBMS we use does not directly report resource metrics per-tenant. The database process provides a measure of the number of disk read requests issued to the OS. However, due to the OS page cache, an actual disk access happens only after a miss in the OS page cache. Our first approximation was that every access to the page cache has a uniform probability of a miss. This approach to modeling the OS page cache is inaccurate since it overlooks cache impedance in the page cache level. An artifact of using a DBMS that utilizes the page cache and does not use direct I/O. If a tenant D_1 misses the database cache more frequently compared to another tenant D_2, D_1 issues more requests to the page cache compared to D_2, thus dominating D_2 in the page cache. Therefore, a request by D_2 has a higher likelihood of a miss compared to that of D_1. The probability of a page cache miss is also dependent on other factors such as the tenant's database size, what fraction is cached, and the page cache eviction policy.

Let $P(A)$ be the probability that an access to a page missed the buffer pool. If h is the cache hit ratio, then $P(A) = 1 - h$. Let $P(B)$ be the probability of a miss in the page cache. If p is the

total number of database pages for the tenant and m is the number of the tenant's pages in the page cache, then $P(B) = 1 - m/p$. The probability of a page being read from the disk is $P(A \cap B)$. For simplicity, if we assume that A and B are independent, then $P(A \cap B) = P(A)P(B) = (1-h)(1-m/p)$. The number of pages accessed per second is given as the product of the operation complexity (o) and throughput (t). Writes contribute to disk activity due to dirty buffer pages, and WAL writes. Update operations provide fixed disk-write activity due to logging, and have a probability of creating another disk-write, if a clean buffer page is dirtied. Outside of logging, updates to an already dirty buffer page may not force a new disk-write. This results in disk activity being difficult to accurately model. Let u be the update operations per second, d be the percentage of pages that are dirty in the buffer pool, and α be a slack variable for updates causing additional disk writes. Therefore, we have **Expected Disk IOPS** :

$$(o \times t)(1-h)(1-m/p) + u + \alpha(u \times d \times (1-h)) \quad (1)$$

This measure (1) is an approximation since it simplifies the effect of updates on disk-writes and assumes buffer pool misses and page cache misses are independent. However, our experiments in Section 5.2.1 reveal that (1) is close enough as a guideline for labeling the training set. We provide all the performance measures to the tenant classifier that *learns* the function using the attributes, but use the transformed expected disk IOPS in generating labels for our training set.

Training the tenant model requires minimal guidance from a human administrator. An administrator analyzes the distribution of the values along the dimensions `D`, `T`, and `O` to determine the boundaries for the buckets that form the class labels for training. For our evaluation, we derived boundaries by examining the distributions of attributes against server resource consumption when tenants were run in isolation. Once the bucket-boundaries are determined, the administrator assigns labels to the tenants based on their respective performance features. Pythia trains a classifier on the labeled training set to learn the tenant model $\mathbb{T}$.

Pythia is designed to work with various tenant models, as long as a model label exists that captures the multiple dimensions of resource utilization. Modeling database interaction in a multitenant environment is challenging due to shifting bottlenecks and unforeseen interactions. This work focuses on demonstrating the potential of a framework that relies on machine learning to model tenant behavior and predict interaction. The tenant model presented in this paper is a representative example. For future work, we plan to examine additional tenant models, such as online models based on query analysis [18], providing administrators feedback on models, and assisting tenant modeling through unsupervised learning.

3.3 Node Model for Tenant Packing

Pythia uses the tenant model to learn which tenant classes perform well together and which do not. The goodness of a packing depends on the class of tenants that comprise the packing. A node model is trained for a single hardware configuration.

A set of tenants at a node is represented as the **packing vector**. If $|\mathcal{C}|$ is the number of tenant classes, then a tenant packing is represented by a vector of length $|\mathcal{C}|$. A position in the vector represents a tenant class and the number of tenants of that class contained in the packing; if a type is absent in the packing, the corresponding count is set to 0. For example, if c_1, c_2, and c_3 are the known tenant classes and a packing had two tenants of type c_1 and three tenants of type c_3, the vector for the packing is $[2, 0, 3]$. The **node feature** representing a packing at a node is the packing vector.

We train the node model ($\mathbb{P}$) by providing a set of labels ($\mathcal{G}$) representing the goodness of a packing. In its simplest form, $\mathcal{G}$ can be $\{\checkmark^+, \checkmark, \times\}$ representing good or bad packings respectively. A packing is good ($\checkmark$) if all tenants meet their SLOs and under($\checkmark^+$) if SLOs are met and server resources are under-utilized. A latency SLOs is composed of an upper bound time for a given percentile. Let $\mathcal{S}$ be the set latencies, composed of s_i, the latency limit for the ith percentile in milliseconds. For our evaluation we set $\mathcal{S} = \{s_{95} : 500, s_{99} : 2000\}$ based on discussions with several cloud service providers. Relaxing or tightening SLOs depends entirely on the applications using the service.

The binary labeling technique captures the SLOs but does not consider utilization of the node. For instance, a packing might be good but the node's resources might be under-utilized or over-utilized. We therefore augment $\mathcal{G}$ to include information about utilization. As noted earlier, disk IOPS and CPU are two critical resources. We use idle CPU percent and the percentage CPU cycles spent on IO (IOWait) as indicators of node utilization both in terms of CPU and disk bandwidth; too many disk requests are reflected in high IOWait. A node's utilization is subdivided into three categories: if idle is above a certain upper bound ($\mathcal{U}_u$) then the node is under-utilized (`Under`), if idle is below a lower bound ($\mathcal{U}_l$) or IOWait is over a threshold $\mathcal{U}_w$ then the node is over-utilized (`Over`), idle percent in the range $(\mathcal{U}_l, \mathcal{U}_u]$ and IOWait less than $\mathcal{U}_w$ is considered good utilization (`Good`). If any tenant violates a latency SLO the node is labeled as Over, regardless of resource consumption. Composing the utilization based division with the SLO based division results in a set of labels that captures both utilization and SLOs: $\mathcal{G} = \{$`Under(`$\checkmark^+$`)`, `Good(`$\checkmark$`)`, `Over(`$\times$`)`$\}$.

To train the node models, a human administrator specifies the parameters $\mathcal{S}$, $\mathcal{U}_l$, $\mathcal{U}_u$, and $\mathcal{U}_w$. A simple rule-based logic assigns labels to the node based on the node feature. Once the training set is labeled, Pythia trains a classifier to learn the node model $\mathbb{P}$.. The node model is incrementally updated to reflect new observations in the running system.

3.4 Utilizing Machine Learning

We use Weka, an open-source machine learning library, to train Pythia's models. We experimented with multiple classifiers such as decision trees, random forests, support vector machines, and classifiers based on regression [23]. The training data is obtained by augmenting an operational system, for which Pythia will be trained, to collect the tenant and node features which are then labeled as described earlier. In our evaluation Pythia utilizes Random Forests, an ensemble decision tree classifier, due to its high accuracy and resistance to overfitting. Once the tenant and node models are trained, they are stored and served in-memory; Delphi uses these models for intelligent tenant placement.

In this section, we presented one way of training the models in Pythia as a representative example. However, Pythia can be adapted to work with a different set of performance features, tenant and node labels, and semantics associated with the label. The role of a domain expert or a system administrator is to determine representative features and assign labels so that the tenant and node models can be trained accordingly. For example, disk I/O is limited by the underlying hardware and experienced administrators can easily identify and categorize ranges of disk consumption. Moreover, Pythia can also be extended to use other forms of machine learning such as clustering or regression learning [23]. Exploring such directions are possible directions of future work; our initial focus was to leverage classification with domain knowledge to explore the end-to-end design space.

4. DELPHI IMPLEMENTATION

We implemented Delphi on a multitenant DBMS with each node running Postgres, an open-source RDBMS engine.[2] All the database level performance features ($\mathbb{F}$) are obtained using two Postgres extensions and without any modification to the Postgres code. Early in our project, we also explored MySQL and found the majority of performance measures comprising $\mathbb{F}$ are available through a third-party MySQL extension ExtSQL.[3] In this section, we explain Delphi's components other than Pythia, i.e., the statistics gathering component, and the crisis detection and mitigation component.

4.1 Statistics Collection

Each DBMS node has an agent which interfaces with Delphi. The agent collects the tenants' performance statistics by querying the database process. In our prototype, tenants share the same Postgres instance and have independent schemas (or databases, in Postgres terminology). To gather the performance statistics, we use two extensions to Postgres in addition to Postgres' internal statistics. The extension `pg_buffercache` provides detailed information about the state of the database buffer by table, and the extension `pgfincore` peeks into the OS's page cache to determine which parts of a tenant's database is cached by the OS. Both extensions expose the statistics as a table which the agent queries using SQL issued through a local JDBC connection. A number of queries are issued to Postgres, `pg_buffercache`, and `pgfincore` to obtain statistics such as per-tenant database and OS page cache allocation, cache-hit rations, number of dirty pages, and read/write ratios.

The agent also collects aggregate node-level usage statistics such as percentage idle CPU, CPU cycles blocked on I/O calls, CPU clock speed and number of cores, memory usage, number of disk blocks read or written, and disk I/O operations per second for all drives hosting database files or the transaction log. The agent can also be configured to follow database logs to record database events such as a checkpoint initialization or slow queries.

The statistics collector requests a snapshot via the agent's web server which obtains the performance measures from the local database; statistics are reset after collection. The response by the agent wraps all the statistics and the time since the last report in a flexible interchange format, JSON in our case, allowing easy extensibility. This entire process takes on the order of few milliseconds and allows lightweight statistics collection. The impact of monitoring on database latency was observed to be a few milliseconds.

4.2 Crisis Detection and Mitigation

4.2.1 Monitoring and Crisis Detection

The statistics collector periodically gathers statistics from all the DBMS nodes to create an aggregate view of the system. For every incoming snapshot, Delphi uses Pythia's tenant model to determine each tenant's class. Delphi maintains a per-tenant sliding window of the last $\mathbb{W}$ snapshots; all Delphi's actions are based on $\mathbb{W}$. The class labels in $\mathbb{W}$ are used to determine a representative label for each tenant. For instance, assume Delphi maintains 5 snapshots, and in this window a tenant D_i has 4 labels corresponding to class c_j and 1 label corresponding to c_k. Delphi represents D_i as $\{0.8c_j, 0.2c_k\}$, i.e., D_i is of type c_j with confidence 80% and type c_k with confidence 20%.

Delphi's use of a window $\mathbb{W}$, rather than using the last snapshot, provides a more confident view about shifts in behavior. It allows Delphi to filter spurious behavior, such as sudden spikes in activity or higher than average response times resulting from system maintenance activities such as checkpoints. Using percentile latency SLOs also limits the impact of a few queries with high latency. Crisis mitigation steps, such as migrating some tenants out of a node, are expensive and hence Delphi must filter out spurious behavior and react only when a shifting trend is observed.

For a given tenant packing P_j at node N_j, Delphi determines whether all tenants' SLOs, $\mathcal{S}$, are being met. If all SLOs are met and no resource is being over-utilized, then this packing is an instance of a good packing for the node model. The slack in resource consumption determines the aggressiveness of consolidation. If one or more SLOs are violated, then this packing is an instance of a bad packing. Once a performance crisis corresponding to a bad packing is detected, Delphi searches for a good packing and takes the remedial measures.

The node model $\mathbb{P}$ receives continuous feedback about good and bad tenant packings and is incrementally updated as Delphi observes new packings and their outcomes. We incrementally re-train $\mathbb{P}$ using the negative examples, i.e., cases where the model's prediction was inaccurate, and a sampling of positive examples that are not repetitive, as many packings and their outcomes are repetitive during steady state.

4.2.2 Crisis Mitigation

Mitigating a performance crisis for a bad packing P_j at node N_j entails identifying a packing $P'_j \subset P_j$ such that $\mathbb{P}(P'_j) \not\equiv \times$, where $\times$ corresponds to an over packing according to the node model $\mathbb{P}$. We formulate this problem of finding the packing P'_j as a search problem through the combinations of the tenant packings, a well-studied problem in artificial intelligence [17]. The search algorithm performs *what-if* analysis using the tenant model $\mathbb{P}$ to determine potential destinations node that can accommodate a subset of tenants in P_j^M without itself deteriorating to a bad packing as predicted by $\mathbb{P}$. Once a good packing P'_j is determined, the tenants $P_j^M = (P_j - P'_j)$ must be migrated out of N_j. Pythia must find one or more destination nodes that can accommodate P_j^M. Therefore any tenant in $p_j^M \in P_j^M$ requires a destination node N_k serving a packing P_k such that $\mathbb{P}(P_k) \not\equiv \times \wedge \mathbb{P}(P'_k) \not\equiv \times$ where $P'_k = P_k \cup p_j^M$. Destinations must be found for all tenants included in P_j^M.

We implemented a few different search algorithms in designing Pythia. *Breadth first search* (BFS) [17] first tries all combinations of migrating one tenant, then combinations of a pair of tenants, and so on until it finds a good packing according to $\mathbb{P}$. Using an exhaustive search algorithm would often not converge on a solution, either due to the search space complexity or not being able to satisfy a goal test of having no nodes in violation. Additionally, we do not expect BFS to scale with a large number of tenants and nodes. The local search algorithm, *hill-climbing* becomes the natural selection, due to the ability to provide a time-bounded best solution, and the ability to treat packing as an optimization problem [17]. With hill-climbing, all immediate neighbors (potential migrations) are examined, and the move providing the largest improvement is selected. Therefore only the local state is considered when making search decisions. This process is repeated, until no additional step can improve the state (a local maximum) or time has expired. Each step is evaluated with a heuristic cost estimate h to find a migration which provides the largest improvement to the tenant packing, by finding a local minimum for h. The naive cost function h attempts to minimize the number of nodes which are labeled as over($\times$).

$$h = \sum_{\{N_i \in N | \mathbb{P}(P_i) \equiv \times\}} 1 \quad (2)$$

[2] `http://www.postgresql.org/`

[3] MySQL: `http://www.mysql.com/`, ExtSQL: `http://www.extsql.com/`.

However, in packings with minimal excess capacity, this cost function (2) to minimize the number of over nodes, would simply overload one node to the point of being unresponsive. The next step would be to minimize the number of tenants in violation. This requires we extend the node model to provide a confidence score λ for a given label. In place of a single label, most classifiers can produce a set of labels and confidences. We denote a function that provides a confidence rating given a packing and label, as $\lambda(\vec{\mathcal{C}}, \mathcal{G}) \rightarrow \mathbb{R} \in [0, 1]$. The new cost function follows as:

$$h = \sum_{N_i \in N} (\lambda(P_i, \times) \cdot |P_i|)^2 \quad (3)$$

This cost function has an artifact of migrating tenants between nodes that were not in violation, to reduce the overall score. Healthy nodes with a large number of nodes, can be labeled as having a small confidence of being over($\times$). This cost function would favor migrating from a large number of tenants with a low over-score, rather than a small number of tenants, in violation, with a higher over-score. We use a minimal threshold, σ for λ to register a score. After examining latencies and IO wait times for $\lambda(\vec{\mathcal{C}}, \times)$ we settled on $\sigma = 0.35$, due to reduced likelihood of high latencies or strained I/O. These results are included in Section 5.2.2. Our final cost function is set to:

$$h = \sum_{\{N_i \in N | \lambda(P_i, \times) > \sigma\}} (\lambda(P_i, \times) \cdot |P_i|)^2 \quad (4)$$

In case the search algorithm cannot find a suitable packing, or cannot converge after a few iterations, it concludes that new nodes need to be added to accommodate the changing tenant requirements and behaviors. In a cloud infrastructure, such as Amazon EC2, Delphi can automatically add new servers to elastically scale the cluster. In a statically-allocated infrastructure typical in classical enterprises, Delphi flags this event for capacity planning by a human administrator. Since the workloads are dynamic and we employ heuristics to find a solution, a stable state is not guaranteed. Additional heuristics, such as maximum allowed moves in a time period, are used to prevent excessive tenant movement.

Delphi must *migrate* the tenants p_j^M for which it found a potential destination node. This problem of migrating a tenant database in a live system is called **live database migration** [9]. Once a tenant is migrated, the outcome is recorded in the history database. Migrating a tenant incurs some cost, such as a few aborted transactions and an increase in response times. Our current search algorithm does not consider this cost in determining which tenants to migrate. Ideally, Delphi will factor migration cost into decision making, however this requires accurate models to predict migration cost, which is influenced by tenant attributes, colocated workloads, and network congestion. We evaluated regression models based on the attributes in Section 3.1 to predict migration cost, but interference from source and destination workloads resulted in inaccurate models. Accurate models to predict migration cost, augmenting the search algorithm to consider a predicted migration cost, and techniques to recognize patterns in workloads are worthwhile directions of future extensions.

It is possible to use replicas for crisis mitigation and load balancing in Pythia. Leveraging existing replicas to migrate workload instead of data migration reduces the migration cost. Synchronous replication protocols make this operation simple and quick, but at the cost of increased latency for update operations during normal operation. Asynchronous replication could also be used, but automating the process of migrating workload to a secondary replica, when the replicas can potentially be lagging, while preserving data consistency requires careful system design and additional approaches. Moreover, even in a system with multiple replicas, Pythia can help select a candidate replica to migrate the workload to. In scenarios where the existing replicas are not suitable destinations (since they might become overloaded as a result of this workload migration), Pythia can also help select a viable destination where a new replica can be regenerated. Our decision to focus on migration limits the problem's scope. In the future, we plan to explore how Pythia can be extended to support a hybrid of workload and data migration.

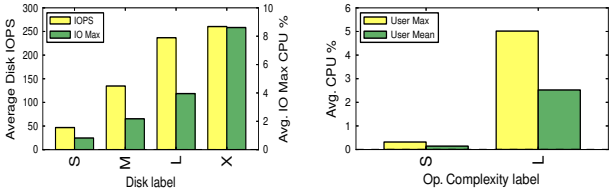
5. EXPERIMENTAL EVALUATION

We deployed and evaluated Delphi on a cluster of 16 servers dedicated to database processes, six servers dedicated to generating client workloads (workers), and one server dedicated to hosting Delphi and a benchmark coordinator. The database servers run PostgreSQL 9.1 on CentOS 6.3, with two quad-core Xeon processors, 32 GB RAM, 1 GB Ethernet, and 3.7 TB striped RAID HDD array. Adhering to PostgreSQL best practice, the buffer pool size is configured to 8 GB, with the remaining memory dedicated to use by the OS page cache. Default settings are used for other configuration options. The following section describes benchmarks used for workload generation, the methodology used to generate various tenant workloads, a validation of the tenant and node models, and a detailed evaluation of Pythia.

5.1 Benchmark and Tenant Description

Delphi targets multitenant environments that serve a wide variety of tenant types and workloads where the tenants often use their database for multiple purposes. Classical database benchmarks, such as the TPC suite, focus on testing the performance and limits of a single high performance DBMS dedicated either for transaction processing (TPC-C) or for data analysis (TPC-H). Existing benchmarks provide little support for evaluating the effects of colocating multiple tenants, systematically generating workloads for large numbers of small tenants, generating correlated and uncorrelated workload patterns, or generating workloads that change or evolve with time. We therefore designed and implemented a custom framework to evaluate and benchmark multitenant DBMSs.

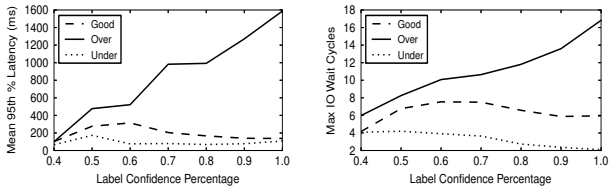
Our multitenant benchmark is capable of generating a wide variety of workloads, such as lightweight tenants with minimal load and resource consumption, a mix of reporting and transactional workloads, and workloads that change behavior with time, thus emulating the variety of tenants a multitenant DBMS can host [2]. The benchmark is a distributed set of **load generator workers** orchestrated by a **master**. The core of the load generator comprises a set of configurable predefined workload types. Our current implementation supports the following workload classes: a light workload of short transactions, composed of 1-4 read and write operations; a lightweight market-based web application which tracks frequent clicks, reads products for browsing, places transactional orders, and reports on related items and popular ads; a time series database with a heavy insert workload and periodic reporting; a YCSB-like [5] workload on larger databases with 80% of operations, on 20% of the data; and a set of YCSB-like workloads with bounded random configurations. A tenant's workload is specified as one workload type, a database size varying between 100MB to 14 GB, and a vector of randomized configuration parameters, including throughput and number of client threads. Therefore a tenant's workload can potentially comprise multiple combinations of workload configurations in different time intervals. Using the randomized configuration, 350 tenants were generated and associated with a random id. Each tenant was run in isolation for a warm-up period of at least 30 minutes, and then latencies were measured over 10 minutes. To ensure tenants are amenable to consolidation with our latency SLOs, tenants with latency 95th percentile greater than 500 milliseconds (ms) or 99th percentile greater than 2000 ms are re-



(a) Expected disk IOPS and CPU I/O Wait

(b) Operation Complexity and CPU Utilization

Figure 4: Tenant model resource consumption when run in isolation.



(a) Average tenant 95th percentile latency.

(b) Average server max CPU cycles blocked on I/O.

Figure 5: Node model performance by label confidence

moved from the set of candidate tenants, which left 314 tenants. While this workload combination does not encapsulate the variety of workloads encountered in a real database platform, it is more robust than using a combination of heterogeneous workloads, such as only using YCSB or TPC-C for all tenants.

The benchmark master receives a mapping of tenants-to-servers from Delphi, and executes all workloads by distributing workloads at the thread granularity in round-robin to all worker nodes. Worker nodes connect directly to database servers via JDBC. Periodically the master collects rolling logs of all latencies, tagged by operation type, workload class, and tenant ID. Implementing such a distributed load testing environment is necessary to generate enough client load to stress 16 database servers concurrently, aggregate usage statistics, and emulate a distributed tenant query router.

5.2 Model Evaluation

Before describing the experimental evaluation of Pythia, we briefly revisit model generation and provide a basic validation that the proposed models capture behavior that Pythia utilizes for placement decisions. Pythia's models must be trained before Delphi can leverage Pythia for managing the tenants. Training data is collected from the operational system and a human administrator provides rules to label the training data, which is then fed into Pythia to learn tenant and node models. A model's accuracy is computed as the percentage of occurrences where Pythia's predicted tenant or node label matches that provided by the administrator. To measure accuracy, we used cross-validation, where the labeled data is partitioned into a *training set* and a *validation set*, the models are trained on the training set and tested on the validation set. Accuracy of the tenant model was about 92% while that of the node accuracy was about 86%. We also validated that when a node's resources are not thrashing, then a tenant's class label is static if the tenant's workload is static. When resource capacity becomes constrained, due to migration or thrashing, we did observe that tenant labels do fluctuate. Labeling based on a sliding window can reduce fluctuation, but fluctuations could result in a misclassification of a node, and requires additional iterations to resolve. Improving tenant interference modeling is left for future work.

Delphi is an initial step in building a multitenant controller, and to focus the problem on tenant placement we currently do not factor the cost of migration in placement decisions. Therefore in all evaluations, when any new combination of colocated tenants is evaluated (a **run**), the framework runs a staggered half-hour warm up period followed by a statistic reset. After warm up, evaluations run for a given time period with snapshots recorded every five minutes and node statistics captured every 30 seconds.

5.2.1 Tenant Model Evaluation

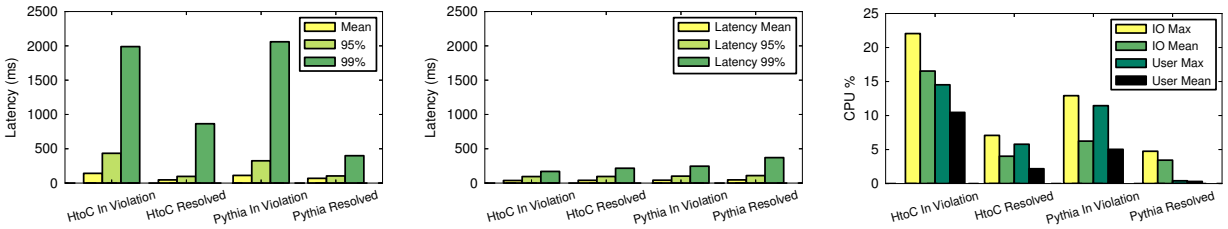
In determining which generated tenants are amenable to the defined latency SLOs, all tenants are run in isolation on a database server. Here the tenants' performance features are labeled using the rules described in Section 3.2. The labeled data set serves as our initial training set for tenant models. To validate that the resource-based tenant models do capture relative resource consumption, we examine the tenant models when run in isolation. The models are labeled using only the database features captured by an agent, and the servers resource utilization is only used for validation. Figure 4 shows that our models are representative of actual resource usage, without direct monitoring. Disk activity is examined in Figure 4(a) where we compare label buckets of **s**mall, **m**edium, **l**arge, and e**x**tra-large against average disk IOPS and the max CPU cycles blocked on I/O (**IO Max**). As predicted the disk component of a tenant's label corresponds to the average observed disk activity. Figure 4(b) shows that average operation complexity (number of pages accessed per transaction), which translates into CPU consumption. Our hypothesis is that high operation complexity includes CPU intensive queries, including, reporting queries that access many pages, complex join operations, or long running transactions that require concurrency validation. Here we show labels with **s**mall and **l**arge operation complexity against mean and max CPU cycles used on user processes, which is primarily composed of the database processes. The range for CPU cycles appear low, but these percentages are across 16 OS threads. Servers with fewer cores, would exhibit higher percentages.

To experiment with the robustness of the tenant model, we ran a TPC-C like tenant without having any TPC-C workloads in the training set. We ran the workload with five warehouses and a throttled single terminal. As expected the model labeled the tenant `DM-TS-OL`, as having medium disk access, low throughput and large operational complexity due to complex transactions.

5.2.2 Node Model Evaluation

In contrast to the tenant model, labeling the node training data can be substantially automated. The input to the node model is a vector of tenant model counts that are colocated on this node. The training requires observing many combinations of colocated tenant workloads. An administrator sets the parameters for determining a node's health, by determining acceptable ranges of resource consumption, such as disk IOPS or CPU consumption, and the percentile-based latency response time SLOs. With the model parameters defined and the ability to run synthetic workloads, Delphi is able to the automatically build the node model. Node models are valid for one type of hardware configuration.

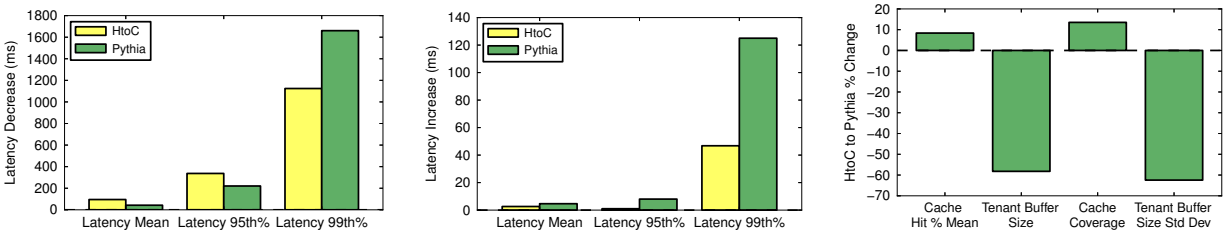
Figure 5 presents average resource consumption by the node's label and Pythia's confidence of the provided label. Figure 5(a) shows the average tenant 95th percentile latency, and Figure 5(b), the average maximum percentage of CPU cycles blocked on I/O (**IO Wait**). The distribution for CPU utilization is similar to I/O, but with a sharp plateau for over labels. The corresponding graph is omitted for space. These results demonstrate that as Pythia becomes more confident about a predicted node label, the results trend



(a) Median latency for nodes experiencing SLO violations, before and after load-balancing.

(b) Median latency for nodes not initially experiencing SLO violations, before and after load-balancing.

(c) Change in CPU utilization for nodes in violation.



(d) Median latency decrease after crisis mitigation for nodes experiencing crisis.

(e) Median latency increase after crisis mitigation for nodes not initially in crisis.

(f) Relative differences from HtoC to Pythia's resolved state.

Figure 6: Comparing improvements to nodes in violation, and the impact on nodes not in violation

towards expected behavior. For example, as a node label increases in confidence of being over, we observe latencies, CPU utilization, and cycles blocked on I/O spike. As a node label becomes more confidently under, latencies, CPU utilization, and blocked I/O cycles decrease. These results imply that Pythia is able to predict the expected resource consumption for tenants. These figures include data collected from the experiments described in Section 5.3.

5.3 Crisis Mitigation

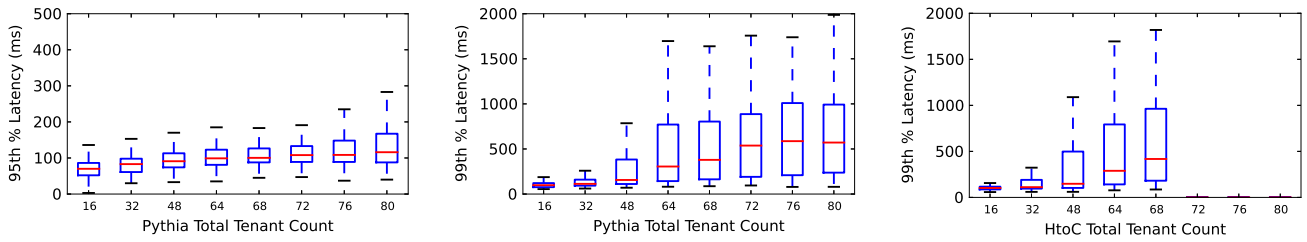
To evaluate Pythia's effectiveness in mitigating a performance crisis, we provide a random tenant packing with a set of nodes in violation, and initiate load-balancing to resolve the crisis. We then iteratively add new tenants to the system, which can result in new tenant violations. If any step does not contain a violation, we continue to add tenants to trigger a violation. The process is repeated until a violation cannot be resolved. We compare Pythia to a greedy baseline load-balancing algorithm **Hottest-to-Coldest (HtoC)**.

HtoC is modeled on the greedy load-balancing baseline used in the evaluation of the large scale storage system, Ursa [25]. This algorithm attempts to iteratively balance load, by moving tenants from over-loaded (hot) nodes to under-loaded (cold) nodes. Faced with a violation, non-violating nodes are inserted into a queue of possible destinations for violating tenants. The queue is sorted in descending order for excess CPU capacity (**Idle CPU**). Idle CPU capacity is a natural single metric to use for resource capacity, as non-idle cycles include cycles for the database process, kernel usage, and CPU cycles blocked on I/O. HtoC iterates through violating nodes by lowest idle cycles, and migrates one random tenant to a node removed from the head of the destination queue. This process repeats until a solution can be found, or until a maximum iteration count is reached and no solution is found. We compared moving a random tenant with moving all violating tenants, and found that a random tenant resulted in fewer violations, had lower average latencies, and resolved crisis with fewer iterations.

For this evaluation we initially assign a small uniform number (two or three) of tenants to all servers and iteratively run the following steps. Initially, all tenants are warmed up for thirty minutes, agent statistics are reset, and then collect a snapshot after running all tenants for five minutes. Delphi checks if any tenant is experiencing a performance crisis with any tenant violating performance SLOs. If no node is in violation, we distribute one new random tenant per server and repeat, starting with a new warm up. If any node is in violation, we attempt to mitigate the crisis by balancing the load through tenant migration. After the tenant repacking is executed, a snapshot is measured after warm up. This re-balancing is allowed to repeat for six iterations, if the re-packing cannot converge by then the experiment interval ends. We alternate complete incremental packing runs between Pythia and our greedy baseline load-balancing HtoC, giving both algorithms an identical list of tenants to use. We start with a low number of initial tenants, and allow each load-balancing algorithm to pack tenants incrementally to avoid using an arbitrary dense starting point that may favor one solution. This also allows both algorithms to be evaluated in light and dense tenant packings.

Because Pythia is more judicious with tenant packings, we can also use this experiment to evaluate the ability for tenant packing, or consolidation. Throughout all of these experiments, HtoC was never able to pack more tenants than Pythia. On average Pythia was able to successfully pack 71 tenants, a 45% improvement over HtoC's 49 tenants. The maximum number of tenants packed using Pythia was 80, and 64 for HtoC. We expect a larger number of tenants could have been packed for both algorithms if smaller tenants were used, durability settings were relaxed, or an array of SSDs were used. On successful load-balances, Pythia converged after 1.75 iterations, where HtoC resolved after 2.25 iterations on average. Pythia migrated an average 5.25 tenants per round, and HtoC migrated 2.25 tenants per round. One reason for increased tenants migrated is that Pythia would often shuffle tenants between non-violating nodes, in order to make capacity on ideal destinations for tenants from violating nodes.

Figure 6 demonstrates Pythia and HtoC's ability to mitigate a crisis by examining the impact of load balancing on tenant latency and resource consumption. The data here is captured from all incremental growth rounds that are successfully load-balanced. Fig-



(a) Pythia 95th percentile (b) Pythia 99th percentile (c) HtoC 99th percentile

Figure 7: Tenant latencies by platform total tenant count.

ure 6(a) shows the before and after median tenant latencies hosted on nodes experiencing a performance crisis. Figure 6(d) shows the same data, but only the decrease in median latencies from load-balancing, instead of the before and after latencies. It is important to note, while HtoC and Pythia exhibit comparable performance gains for mean and 95th percentile latencies, Pythia decreases the 99th percentile latency by about 50% more, despite hosting 45% more tenants on average. Figures 6(b) and 6(e) show the impact of load-balancing on nodes not experiencing a SLO violation. As expected, Pythia has a larger impact on latency due to the increased number of tenants migrated from violating nodes. For both approaches, the increase in latency is small compared to the decrease in latency of violating nodes. Again, we assume that any database hosted in a multitenant environment can tolerate some variance in latency, provided latency SLOs are met. While violating latencies are similar for both approaches, Figure 6(c) shows HtoC's violating resource consumption is substantially worse than Pythia. Additionally, the resolved state for Pythia has lower CPU usage than non-violating nodes(not depicted) by 1% point on average.

A goal of load-balancing with Pythia is to implicitly consider cache impedance when matching tenants, in order to provide tenants with adequate cache access to meet latency SLOs. Figure 6(f) compares the relative differences in resource attributes between the resolved states of Pythia and HtoC. Pythia's resolved state results in a higher average tenant cache hit ratio and a higher percentage of the database that is in the page cache and buffer pool (**cache coverage**), which results in reduced disk activity for the tenants. An improvement of 13% on cache coverage and 8% on cache hit ratio alleviates a substantial amount of disk seeks, thus reducing disk contention between tenants. Interestingly, after Pythia mitigates a performance crisis, the tenants remaining on previously violating nodes have a higher cache hit ratio, a smaller average buffer size, and a lower variation in buffer size, when compared with the resolved HtoC nodes. The higher cache hit ratio combined with smaller buffer size suggests that tenants with smaller working sets remain together. The reduced variation, measured by standard deviation, means that the buffer size is more uniform, each tenant is getting a relatively equal share of the buffer pool, and that the cache is not cannibalized by dominate tenants. We therefore, conclude that Pythia is matching tenants' cache impedance when selecting the ideal packing to resolve a crisis.

To gather additional insight into the packing limits of both algorithms, we selected two successful packings of 64 tenants to grow at a smaller rate. We repeatedly grew these packings for both algorithms, by adding one tenant to four random nodes in each growth round. Pythia was able to successfully pack 72, 76, and 80 tenants; HtoC could not successfully pack beyond 68 tenants. Figure 7 shows latency distributions by total tenant count, for all successful growth rounds for both experiments described in this section. The boxplots show sampled minimum, lower quartile, median, upper quartile, and sampled maximum percentile latencies. The 95th percentile latencies for HtoC are very similar to Pythia's, so this graph is omitted. As we can see from Figure 7 the 99th percentile latencies are a primary driver for SLO violations, in both Pythia and HtoC. The sampled maximum and upper quartile latencies for HtoC rise much faster than Pythia, resulting in violations from fewer tenants. Our belief is that Pythia is optimizing packings for the 99th percentile, as most packing violations result in violations for the 99th percentile. This is a likely reason why this latency category has the biggest gains for Pythia. Experimenting with a 99th percentile latency of 10,000 ms and a 95th percentile of 500 ms, Pythia successfully packed 88 tenants, while HtoC could still not pack more than 68 tenants due to 95th percentile violations.

6. RELATED WORK

A vast corpus of research exists for problems related to Delphi's design. This includes database multitenancy, consolidation, resource orchestration, performance modeling, and crisis mitigation. Section 1.1 discusses the related work in managing multitenant DBMSs [1, 6, 14, 24]. Issues related to the design of multitenant systems are covered in Section 2.1 [3, 7, 13, 22].

Modeling resource consumption for application placement and load balancing has been successfully applied to many domains. Modeling performance of n-tiered web services has proven effective for predicting response time [20]. Regression based analytic modeling for n-tier applications has been used for capacity planning and placement, by approximating CPU demand [26]. These analytic models do not translate well for multitenant DBMS environments, due to dynamic and ad-hoc usage, and high latencies from shifting bottlenecks due to multivariate resource contention [1].

Storage systems are another domain which has utilized modeling for placement. Pesto, builds sampling-based models to understand workload performance, and load-balance I/O utilization [11]. The models are based on a linear relationship between outstanding I/O requests and latency, an assumption that does not hold for DBMSs. Similar to Pythia, Romano builds statistical models to predict workload performance, and to predict the impact of colocated workloads [16]. Load-balancing in Romano uses Simulated Annealing to avoid local maximums when finding new packing plans, and a greedy approach to minimize migrations when implementing the re-packing. Both Romano's interference models and search optimizations could extend Pythia in future work. Ursa, targets large scale storage systems by optimizing load balancing to only mitigate hot-spots and leverage topology-aware migrations to minimize re-packing costs [25]. This approach can be leveraged when migration costs are factored into Pythia.

Beyond provisioning and orchestration, Pythia's tenant model can be extended to better understand usage patterns. This can include characterizing service spikes [4], or predicting shifts in workloads [12]. Models for predicting storage consumption could ex-

tend disk-activity estimation of the tenant model [15]. In making placement decisions, additional tenant and workload combination models could improve interference prediction, when changing resource consumption limits resource-based models [8, 18].

7. CONCLUSION

Multitenant DBMSs consolidate large numbers of tenants with unpredictable and dynamic behavior. Designing a self-managing controller for such a system faces multiple challenges such as characterizing tenants, reducing the impact of colocation, adapting to changes in behavior, and detecting and mitigating a performance crisis. The complex interplay among tenants, DBMS, and the OS, as well as aggregated resource consumption measures make the task of monitoring and load balancing difficult. We designed and implemented Delphi, a self-managing controller for a multitenant DBMS that monitors and models tenant behavior, ensures latency SLOs, and mitigates performance crises without requiring major modifications to existing systems. Delphi leverages Pythia, a technique to classify tenant behavior, and learn good tenant packings. Pythia does not make assumptions about the tenant workloads or the underlying DBMS and OS implementations. Our analysis revealed unexpected interactions arising from tenant colocation and identified tenant behavior that are most sensitive to resource starvation. Our experiments, using a variety of tenant types and workloads, demonstrated that Pythia can learn a tenant's behavior with more than 92% accuracy and learn quality of packings with more than 86% accuracy. Using Pythia, Delphi can mitigate a performance crisis by selectively migrating tenants to improve 99th percentile response times by 80%.

Acknowledgments

The authors thank Neil Conway, Ken Salem, Jon Walker, Zhengkui Wang, Jerry Zheng, and the anonymous reviewers for providing useful feedback. This work is partly funded by NSF grants III 1018637, IIS 0905084, CNS 1053594, and NEC Labs America.

8. REFERENCES

[1] M. Ahmad and I. T. Bowman. Predicting system performance for multi-tenant database workloads. In *ACM DBTest*, pages 1–6, 2011.

[2] Personal communications with Jerry Zheng, VP Web Operations at AppFolio Inc., April 2011.

[3] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manner, L. Novik, and T. Talius. Adapting Microsoft SQL Server for Cloud Computing. In *ICDE*, pages 1255–1263, 2011.

[4] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC*, pages 241–252, 2010.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154, 2010.

[6] C. Curino, E. Jones, S. Madden, and H. Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *SIGMOD*, 2011.

[7] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *CIDR*, pages 235–240, 2011.

[8] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

[9] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*, pages 301–312, 2011.

[10] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual machine hosting for networked clusters: Building the foundations for 'autonomic' orchestration. In *VTDC*, 2006.

[11] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: online storage performance management in virtualized datacenters. In *SOCC*, pages 19:1–19:14, 2011.

[12] M. Holze, A. Haschimi, and N. Ritter. Towards workload-aware self-management: Predicting significant workload shifts. In *ICDE Workshops*, pages 111 –116, 2010.

[13] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. In *BTW*, pages 514–521, 2007.

[14] W. Lang, S. Shankar, J. Patel, and A. Kalhan. Towards multi-tenant performance slos. In *ICDE*, pages 702–713, 2012.

[15] O. Ozmen, K. Salem, M. Uysal, and M. H. S. Attar. Storage workload estimation for database management systems. In *SIGMOD*, pages 377–388, 2007.

[16] N. Park, I. Ahmad, and D. J. Lilja. Romano: autonomous storage management using performance prediction in multi-tenant datacenters. In *SoCC*, pages 21:1–21:14, 2012.

[17] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[18] M. B. Sheikh, U. F. Minhas, O. Z. Khan, A. Aboulnaga, P. Poupart, and D. J. Taylor. A bayesian approach to online performance modeling for database appliances using gaussian models. In *ICAC*, pages 121–130. ACM, 2011.

[19] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic resource allocation for database servers running on virtual storage. In *FAST*, pages 71–84, 2009.

[20] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI*, pages 71–84, 2005.

[21] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *OSDI*, pages 239–254, 2002.

[22] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *SIGMOD*, pages 889–896, 2009.

[23] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Data Management Systems. Morgan Kaufmann Publishers Inc., second edition, 2005.

[24] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigümüs. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE*, pages 87–98, 2011.

[25] G.-w. You, S.-w. Hwang, and N. Jain. Scalable load balancing in cluster storage systems. In *Middleware*, pages 101–122, 2011.

[26] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC*, pages 27–, 2007.