User Disengagement-Oriented Target Enforcement for Multi-Tenant Database Systems

Ning Li
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
ning.li@uta.edu

Hong Jiang
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
hong.jiang@uta.edu

Hao Che
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
hche@cse.uta.edu

Zhijun Wang
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
zhijun.wang@uta.edu

Minh Q. Nguyen
Faculty of IT, Ho Chi Minh City University of Transport
Ho Chi Minh City, Vietnam
nqminh@live.com

Todd Rosenkrantz
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
stoddard.rosenkrantz@mavs.uta.edu

ABSTRACT

Unexpected long query latency of a database system can cause domino effects on all the upstream services and severely degrade end users' experience with unpredicted long waits, resulting in an increasing number of users disengaged with the services and thus leading to a high *user disengagement ratio* (UDR). A high UDR usually translates to reduced revenue for service providers. This paper proposes UTSLO, a UDR-oriented SLO guaranteed system, which enables a database system to support multi-tenant UDR targets in a cost-effective fashion through UDR-oriented capacity planning and dynamic UDR target enforcement. The former aims to estimate the feasibility of UDR targets while the latter dynamically tracks and regulates per-connection query latency distribution needed for accurate UDR target guarantee. In UTSLO, the database service capacity can be fully exploited to efficiently accommodate tenants while minimizing resources required for UDR target guarantee.

CCS CONCEPTS

• **Information systems** → **Autonomous database administration**.

KEYWORDS

database system, multi-tenant, user disengagement ratio

ACM Reference Format:
Ning Li, Hong Jiang, Hao Che, Zhijun Wang, Minh Q. Nguyen, and Todd Rosenkrantz. 2023. User Disengagement-Oriented Target Enforcement for Multi-Tenant Database Systems. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624668

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624668

1 INTRODUCTION

It has become prevalent that various enterprise-class cloud service or e-commerce service providers move their businesses to the cloud with multi-tenancy, where multiple service providers or tenants share the underlying database processing capacity with pre-specified service level objectives (SLO) [54]. Each tenant may desire a high degree of user engagement due to its high correlation with their sales/revenues. According to a recent Akamai study, an extra service delay of 100-ms can reduce conversion rate, a user engagement metric [1], by 7%, which means a noticeable drop of 6% in sales [15] and $3.8 Billion loss of annual revenue for Amazon [5]. Since database systems play a key role in and have been identified as one of the major bottlenecks for multi-tenant data services [23], it becomes imperative to enable databases to support per-tenant quality of service (QoS) guaranteed services.

According to the industry practice, user engagement/disengagement performance metrics are the ultimate measures of QoS for tenants, including churn rate [7], conversion rate [9], bounce rate [6], the reduction rate in query refinement and satisfaction [57]. Although these metrics

[1]Conversion rate, as a user engagement metric, is defined as the percentage of users who engage in desired actions, e.g., purchases [9].

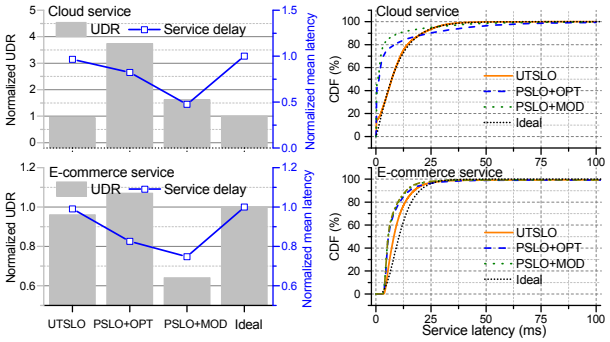

Figure 1: The comparisons in normalized UDR (the one closer to but lower than 1.0 is better), normalized mean latency (the mean latency for the ideal one is 1.0), and CDF curves of service latency for the proposed solution UTSLO and tail-latency-SLO based solutions.

are determined by many factors, such as competitiveness of product prices and qualities, shipping speed and cost, and product returning policies, from the viewpoint of developing e-commerce service platforms or systems, the service response time is probably the most important factor that will impact the user engagement/disengagement rates, as supported by e-commerce service providers [1, 15, 57]. With respect to the database aspect of the system design, it is the database query latency, x, that matters the most. In other words, one can generally define a user disengagement ratio at any given x ($UDRx$) for the tenant connection T_i (measurable in practice), as $u_i(x)$, representing churn rate, non-conversion rate, bounce rate, or reduction rate, etc., at x for T_i. Then the overall user disengagement ratio (UDR), U_i, for T_i, the performance metric that the tenant cares the most, can be generally expressed as, $U_i = \int_{0+}^{+\infty} f_{X_i}(x) * u_i(x)\,\mathrm{d}x$, where $f_{X_i}(x)$ is the probability density function (PDF) of database query latency, x, for queries of the service provided by T_i.

It becomes clear that to provide UDR-oriented QoS, one must be able to control $f_{X_i}(x)$ or equivalently, the cumulative distribution function (CDF) $F_{X_i}(x)$ for individual tenants in order to achieve the desired user engagement/disengagement performance for all tenants sharing the same database. Unfortunately, the state-of-the-art QoS solutions [24, 43, 44, 46, 66] are simply not up to the task, as they all exclusively focus on providing a specific percentile tail latency guarantee (i.e., the p^{th}-percentile of tail-latency $x = x_p$) for applications. More specifically, they all attempt to ensure, $p/100 \leq F_{X_i}(x_p)$, i.e., only need to control $F_{X_i}(x)$ at a single point $x = x_p$, rather than the entire $F_{X_i}(x)$. This makes these tail-latency-SLO based solutions incapable of offering UDR-oriented QoS. Moreover, due to high variability and dynamism of the workload that is inevitable in a consolidated multi-tenant database environment, $F_{X_i}(x)$ may change over time. As a result, a viable solution must involve online control mechanisms that can keep track of and adapt to such changes. This makes the design of a UDR QoS guaranteed solution challenging.

Moreover, multi-tenant database services (e.g., the ones built on developer data platforms or self-managed database service systems such as MongoDB Atlas [19]) provide database *connections* for their tenants' applications (e.g., enterprise-class cloud services or e-commerce services) to share the database query processing capacity. Thus, connections become the basic units of resource allocation within a database system and cannot be directly managed by OS resource management tools (e.g., Cgroups [17]), which can only offer process/thread-level resource management, let alone deploying them in virtual machines or containers. This makes it difficult for most existing multi-tenant QoS solutions [30, 34, 49, 58, 66], which often rely on OS resource scheduling mechanisms, to directly handle connection-level resource allocation for a single database system.

To illustrate the necessity of regulating service latency distribution and the potential issues that existing approaches need to face for the UDR target enforcement, we give a case study for two different scenarios, i.e., enterprise-class cloud services and e-commerce services, under a state-of-the-art any-percentile tail latency SLO guarantee solution PSLO [46] and our solution (i.e., UTSLO) that supports fine-grained database query latency distribution regulation according to UDR targets. The enterprise-class cloud service workload is simulated by Yahoo Cloud Serving Benchmark (YCSB) [25], which establishes 512 connections with a MongoDB [11] key-value store and 10 of them have UDR targets. In contrast, e-commerce services mainly involve Online Transaction Processing (OLTP) workloads generated by Py-TPCC [40], which creates 256 connections with a MongoDB transactional database server and 9 of them belong to UDR-sensitive tenants. To make PSLO workable for UDR targets, we adopt two levels of estimates, i.e., OPT and MOD (elaborated in Section 5), for the unguaranteed percentile range (e.g., (95%, 100%] if a 95^{th} percentile tail latency SLO is ensured) when translating a UDR target into a corresponding tail latency SLO. As shown in Fig. 1, either of the PSLO schemes (based on OPT and MOD) cannot effectively enforce UDR targets and cause severe UDR target violations (e.g., 3.5 times higher and thus worse than the target for enterprise-class cloud services) or over-zealously enforce UDR targets (e.g., only 64% of the UDR target and thus 25% lower mean latency is required for e-commerce services) with serious resource over-provisioning. This is because the UDR measure is determined by the entire latency distribution and hard to be set equivalently at a tail latency target, a single point on the latency CDF curve. In contrast, the proposed solution UTSLO can generate an ideal latency distribution according to the UDR target under highly variable query latency, and dynamically track and regulate latency distribution to approach the ideal one to accurately enforce the UDR target.

As a result, UTSLO can accurately enforce UDR targets for all the US-sensitive connections under both scenarios.

To address the above challenges, we propose a *U*DR *T*arget oriented *SLO* guarantee system (UTSLO) to enforce UDR targets for multiple tenants of the underlying database system (e.g., MongoDB [11]). Specifically, UTSLO adopts a hybrid modeling-and-measurement-based approach to capture the query latency distribution with low cost. It only involves controlling and adapting the relative values of the mean and variance of query latency to maintain the latency distribution exactly meeting the UDR target for each UDR-sensitive (US) tenant's database connection. More specifically, by dynamically regulating the mean latency, or throughput allocation across tenants' connections in response to highly dynamic query latency variation (measured by the coefficient of variation (CV) [8]), UTSLO manages to accurately bound the UDRs for all the US tenants by their respective UDR targets, while maximizing the throughput performance for the best-effort (BE) tenants, whose aim is to share as much throughput as possible. To keep a low-level latency variation, hence, reducing the resources required to meet the UDR targets, UTSLO adopts a UDR-target-aware two-dimensional admission control[2] to simultaneously manipulate multi-tenant query concurrency and per-connection query burst. Thus, in each scheduling round, only the top-C tenants' connections, selected by strategic scheduling policies, are allowed to send a specific number of queries (i.e., quota) to the database system to strike an optimized trade-off between query latency variation and throughput under the UDR constraints.

We implemented UTSLO based on a popular NoSQL database system, i.e., MongoDB [11], to enforce UDR targets by regulating the network-I/O system calls issued by the database for query fetching/processing. In this way, UTSLO can separate its operations from the underlying database processing and thus work with different versions of MongoDB without the need to modify the database system, making it highly flexible and portable. Our extensive evaluation shows that UTSLO can accurately enforce UDR targets for multiple tenants in a cost-effective manner.

2 BACKGROUND AND MOTIVATION

2.1 Background and Case Studies

Multi-tenant database systems allow effective and transparent data storage resource sharing among multiple tenants via their individual database connections. Each of the tenants can provide service to its individual users by its application that typically relies on databases' query processing 24×7[23]. Due to the common dependency between a typical user's commercial behaviors [36] (e.g., searching and comparing before purchasing), the database asynchrony makes it hard to parallelize queries with causal precedence relationships inside a database connection's context and thus the per-connection synchronous query mode is assumed as the best practice to ensure the correct results of users' queries.

① **Database/connection-level multi-tenancy:** One of the great challenges for multi-tenant database systems is how to collocate multiple tenants with adequate performance isolation required by tenants' performance targets. This is even more so for the UDR measure that is determined by the entire service latency distribution, making it harder to guarantee than tail latency SLO. For traditional relational database systems (e.g., MySQL [12], Oracle [13] and PostgreSQL [21], etc.), it is typical to assign each tenant a dedicated database and apply database-level performance QoS solutions (e.g., Delphi [28] for tail latency optimization) to the multi-tenant database management system (DBMS). However, NoSQL database systems such as MongoDB [11] can offer a higher level of availability, scalability, and flexibility by accommodating data from a large number of tenants with a higher query processing capacity than relational multi-tenant DBMSs. Further, it is feasible for MongoDB to support a single database with shared collections for all tenants. In this way, a growing number of tenants can be accommodated with consistent structure of data and query requirements [16]. However, database-level performance QoS solutions cannot be applied to this case. This is because it demands a finer-grained multi-tenant performance QoS solution at the connection level with differentiated target performance levels across tenants' database connections. In this paper, we mainly focus on providing a fine-grained (i.e., connection-level) multi-tenant UDR target guarantee solution without the need to modify database systems. However, our solution can be easily modified for the database-level multi-tenant environment under which each tenant owns its database.

② **Multi-type tenants:** Multi-tenant database systems based on UTSLO can simultaneously serve UDR-sensitive (US) tenants and best-effort (BE) tenants. The former mainly focus on user-facing services and demand their UDR to be below a target in a cost-effective fashion while the latter want as much throughput as possible to drive their business processing. A tenant can own multiple database connections to parallelize query processing for certain QoS levels or throughput enhancement. For instance, an e-commerce tenant can establish three connections for the users visiting its web pages via three different types of end-user devices (i.e., desktop, mobile, and tablet), each of which is characterized by its unique UDR-latency function (or UDRx, as shown in Fig. 2) respectively and required to be treated separately to fulfill their UDR targets. Our solution also covers other types of QoS as its special cases, such as latency or throughput

[2]Admission control strategically imposes limits on multi-tenant concurrent queries and enqueues any additional queries until earlier ones finish without query cancellation, compromising throughput, or causing contention.

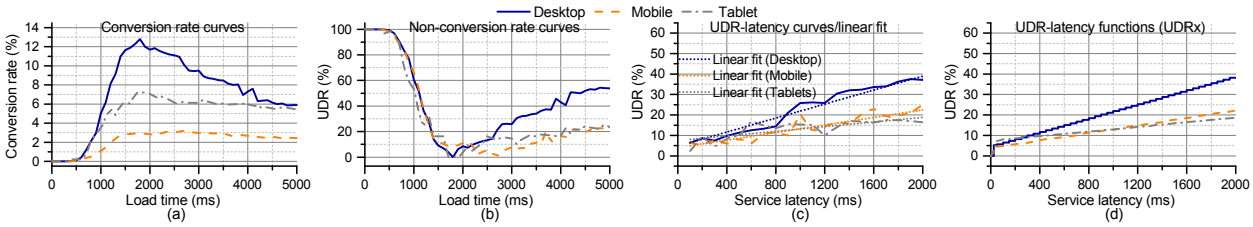

Figure 2: An example that illustrates how to generate UDR-latency functions (UDRx).

SLO, and the cases where US/BE tenants are divided into groups.

③ **UDR-latency function (UDRx):** To evaluate the impact of query latency on UDR based on tenants' individual demands, each US tenant needs to provide its own UDRx (by measurement). Since the measurement must be done in finite latency intervals (e.g., every 100ms) to be tractable, UDRx exhibits a staircase-like curve. To realistically reflect the real-world UDR-latency relationship, this paper focuses on two classic scenarios, i.e., enterprise-class cloud services and e-commerce services, based on the statistics made by enterprises or the report for e-commerce sales by default. For example, according to the survey made by Akamai [2], we can obtain the relationship between conversion rate (a user engagement measure) and load time[3] for e-commerce users via three types of devices (i.e., desktop, mobile, and tablet)[4], as shown in Fig. 2(a). We further translate these conversion rate curves into the relationship between percentage non-conversion rate (a UDR measure) and load time (Fig. 2(b)). Since this paper focuses on the impact of database processing latency on the UDR measure, we assume that other service latency contributing to load time has been optimal, based on which we can get UDR-latency curves by linear fitting (Fig. 2(c)). Note that, based on the studies by Microsoft and Google [57], an extra service latency below a threshold (e.g., 25ms) injected into the overall service delay has unnoticeable impact on UDR measures. We also set a zero-UDR range for database query latency (set at $(0, 25ms]$ by default) and finally get UDRx (Fig. 2(d)).

2.2 Related Work

The existing solutions most relevant to ours would be any-percentile tail latency target guarantee solutions [24, 43, 44, 46, 66]. They can guarantee tail latency SLOs under any predefined percentile, beyond which is an unguaranteed and unpredictable high-percentile latency range. Thus, these approaches are unable to regulate the entire latency distribution, which is required by the UDR target enforcement, since the queries with their latencies falling in the unguaranteed high-percentile range can significantly increase UDR beyond the target and lead to a UDR violation.

[3]The time it takes to fully load a web page.

[4]The faster end of the curves in Fig 2(a) (b) correlate with a lower conversion rate due to non-latency factors (e.g., unavailable web pages/resources).

Another type of tail latency target guarantee solutions focus on the tail latency target enforcement at a specific percentile. For example, Avatar [65] can enforce the 95 percentile tail latency target while Cake [62] enables the 99 percentile's, and the schemes of [31, 38, 67] adopting strict worst-case guarantees essentially focus on the 100 percentile latency objectives. Under the fixed percentile for the latency target, these solutions can only meet the UDR target by roughly estimating the tail latency target, very likely resulting in serious resource under-provisioning/over-provisioning.

There are more solutions that focus on tail latency optimization by means of replica selection and request reissuing [27, 37, 48, 59, 61, 63, 64], congestion control [52, 60] or scheduling techniques[22, 29, 56] for cloud/data-center applications, as well as distribution/correlation-aware optimization techniques [51] or optimal data-passing methods [50] for DAG-based serverless applications. Although these approaches can reduce latency distribution tail by reducing the latencies in a specific percentile range, they cannot guarantee that the resulted latency distribution can indeed meet the requirements of multi-tenant UDR targets.

Existing DBMS QoS solutions/facilities, e.g., Db2 workload manager [10], Oracle Database Resource Manager [18], and the resource governor for Microsoft SQL server [20], etc., can provide different levels of capabilities in planning/prioritizing resource allocation at different levels of the system stack (e.g., session-level and database-level) to apply throughput controls or optimize latency performance. Although these approaches can potentially improve user experience to different extents, they are incapable of offering UDR target guarantees under multi-tenant environments.

In contrast, this paper proposes a UDR guaranteed system UTSLO to precisely enforce per-connection UDR targets for multi-tenant database systems (e.g., MongoDB [11]) with minimized resource over-provisioning. To this end, inspired by the insights obtained from the study cases in Section 1, the design and implementation of UTSLO are to be governed by the principles of ① Estimate the feasibility for the UDR targets by generating and analyzing theoretically optimal query latency CDF curves according to UDR targets for US tenants at the connection level; ② Dynamically track and regulate highly variable query latency distributions as required by UDR targets in a cost-effective fashion.

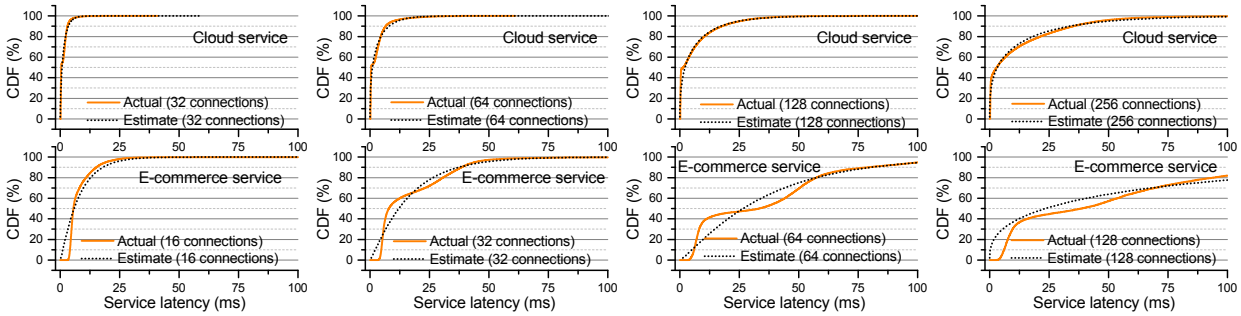

Figure 3: The effectiveness of query latency distribution approximation under different levels of query concurrency.

3 UTSLO DESIGN

In this paper, we propose UTSLO to regulate database connection-level query latency distributions to meet each UDR-sensitive (US) tenant's target in an accurate manner. *In UTSLO, the resources required for the UDR target enforcement will be minimized and thus more tenants (e.g., best-effort (BE) tenants) can be served by the underlying database system.*

Symbols	Description
T_i	The i^{th} tenant's database connection.
U_i	The actual UDR for T_i.
$\hat{U}_i$	The UDR target for T_i.
η_i	The UDR normalized by $\frac{U_i}{\hat{U}_i}$ for T_i.
$[\zeta, \theta]$	The allowed range for η_i.
M_i	The mean latency for T_i.
$\hat{M}_i$	The mean latency budget for T_i.
λ_i	The query throughput for T_i.
$\hat{\lambda}_i$	The query throughput budget for T_i.
CV_i	The coefficient of variation for T_i query latency.
$\hat{cv}$	The upper bound for CV_i.
$u_i(.)$	The UDR-latency function (UDRx) for T_i.
$LR_i(.)$	The relationship between $\hat{\lambda}_i$ and CV_i.
C	The upper bound for query concurrency.
D	The upper bound for query quota.
U^{min}	The min normalized UDR over all US connections.
U^{max}	The max normalized UDR over all US connections.
α	The probability of choosing the US tenants.
γ	The coefficient for reducing α.
β	The coefficient for increasing α.
$\hat{\lambda}_{tot}^{US}$	The total throughput budget for the US tenants.
$\hat{\lambda}_{tot}^{BE}$	The total throughput budget for the BE tenants.
p	The throughput ratio for the US tenants.
$\hat{Q}_i$	The queuing time budget for T_i.

Table 1: Some symbols used in this paper.

3.1 Multi-Tenant UDR Target Guarantee

UTSLO is dedicated to a specific database system (e.g., MongoDB [11]) that supports multiple concurrent *query-intensive* tenants' database connections.

The objective function: For each US tenant's connection $T_i, 1 \le i \le n$, UTSLO aims to minimize the resources required to enforce the given UDR targets (i.e., $\hat{U}_i$) by maximizing its mean latency (i.e., M_i) so that the total query processing capacity provided by the database system (i.e., Ψ) can be fully utilized to serve more tenants, which can be represented as:

$$\begin{aligned} &\text{Maximize } M_i \\ &\textit{subject to} \\ &U_i = \int_{0+}^{+\infty} f_{X_i}(x) * u_i(x)\,\mathrm{d}x \le \hat{U}_i, \forall i, 1 \le i \le n \end{aligned} \tag{1}$$

where $u_i(x)$ is given for T_i while $\hat{U}_i$ has a positive correlation with M_i when the latency variation stays unchanged, i.e., a more stringent (lower) $\hat{U}_i$ typically requires a lower M_i (thus more resources) under a specific level of latency variation represented by CV_i. Moreover, a lower CV_i typically indicates a more stable query latency and thus a shorter tail to make UDR target easier to meet. All these suggests that M_i is maximized, when the UDR targets are exactly met, under the lowest level of latency variation. Hence, UTSLO is designed to dynamically adjust $f_{X_i}(x)$ in response to workload changes to keep all U_i as close to their respective $\hat{U}_i$ as possible, while minimizing the query latency variances with negligible throughput loss.

Query latency distribution: Next, we need to estimate the query latency distribution $f_{X_i}(x)$ for each US tenant's connection. To this end, we adopt the following hybrid modeling-and-measurement approach. We describe the multi-tenant query processing with a single database system by a $G/G/1$ queuing model under heavy load. According to the central limit theorem [53] for heavy traffic queuing systems [41, 42], the distribution of query waiting time can be approximated by an exponential distribution, which can also apply to query latency since the latency distribution will approximate the wait time distribution under a high I/O traffic load [55]. Inspired by this result, T_i's query latency distribution $f_{X_i}(x)$ for any arrival process can be approximated by a generalized exponential distribution function [35]:

$$f_{X_i}(x) = \frac{\phi_i}{\omega_i}(1 - e^{-x/\omega_i})^{\phi_i - 1} e^{-x/\omega_i} \ (x > 0, \phi_i > 0, \omega_i > 0). \tag{2}$$

where ϕ_i and ω_i are shape and scale parameters, respectively. Its CDF function [35] can be represented as:

$$F_{X_i}(x) = (1 - e^{-x/\omega_i})^{\phi_i} \ (x > 0, \phi_i > 0, \omega_i > 0). \tag{3}$$

The corresponding mean and variance are given by [35]:

$$E(X_i) = \omega_i[\psi(\phi_i + 1) - \psi(1)]. \quad (4)$$

$$V(X_i) = \omega_i^2[\psi(1)^{'} - \psi(\phi_i + 1)^{'}]. \quad (5)$$

where $\psi(.)$ and $\psi(.)^{'}$ are the digamma function and its derivative, respectively. Since the distribution given in Eq. 2 or Eq. 3 is determined by the shape and scale parameters (i.e., ϕ_i and ω_i), which can be solved by Eq. 4 and Eq. 5 under the given mean and variance of the query latency, the mean and variance of the query latency completely determine $f_{X_i}(x)$ or $F_{X_i}(x)$. To verify the effectiveness of the approximation, we use the mean and variance of the query latency obtained under different levels of query concurrency (controlled by the total number of database connections) for two type of workloads (i.e., enterprise-class cloud services simulated by YCSB benchmark [25] and e-commerce services simulated by TPC-C workloads [14]) as the input parameters to derive the theoretical latency CDF curves. As shown in Fig. 3, for enterprise-class cloud services, the theoretical latency CDF curves can fit the corresponding actual ones almost perfectly. Although the approximation errors for e-commerce services are higher than those for cloud services, their CDF curves are still very closely approximated especially for higher latencies, which contribute more to UDR.

Intuitively, to keep U_i close to its target $\hat{U_i}$, an increased variance must be compensated by a decreased mean and vice versa. This lends us a convenient way to maintain a UDR, U_i that closely tracks the UDR target for T_i, by regulating the mean query latency or query throughput in response to query latency variability (or variance) changes.

3.2 Mean Latency Budget Exploration

Based on the discussion in Section 3.1, for any US tenant's connection T_i, it is viable to exhaust all the possible combinations of mean query latency and latency variance to find the highest *mean latency budget* or MLB ($\hat{M_i}$) under each specific variance that can determine the latency distribution to exactly meet T_i's UDR target. This will minimize the *throughput budget* (TB) ($\hat{\lambda_i}$) which refers to the query throughput required for the UDR target enforcement when the utilization of the database connection is 100%.

Based on Algorithm 1, we can obtain a suboptimal MLB $\hat{M_i}$ for each tenant's connection T_i under the current measure of latency variation CV_i based on the UDRx $u_i(x)$ and the UDR target $\hat{U_i}$ of T_i by binary searching. Specifically, we set the search scope of query latency for T_i as $(M_{min}, M_{max}]$, where M_{min} can be set at 0 while M_{max} can be determined by the Little's law [47] with the minimum guaranteed throughput per tenant (e.g., 50 ops/s). Before we search an optimal MLB, we first figure out the generalized exponential distribution function (i.e., $F_{X_i}(x)$) based on Eq. 3 by determining its shape and scale parameters (i.e., ϕ_i and ω_i). To this end, we can jointly solve Eq. 4 and Eq. 5 to pin down these two parameters by substituting $\hat{M_i}$ and the latency variance (obtained by $(CV_i * \hat{M_i})^2$) into these two equations, respectively. And then, we can accumulate the UDR contributed by all the latencies indicated in the $u_i(x)$ to obtain the UDR for T_i under the current measurement of query latency variation (CV_i) and the current estimated MLB, which can help to assess the effectiveness and accuracy of the suboptimal MLB.

The criterion of the MLB evaluation is the accuracy of UDR target enforcement, which is quantified by a small range of the normalized UDR (i.e., $\eta_i = \frac{U_i}{\hat{U_i}}$) between ζ and θ. Only if η_i falls in the allowed range, will the resource provisioning for the UDR target enforcement be considered at the minimum level. As shown in Algorithm 1, if η_i is smaller than ζ (0.95 by default), then resource over-provisioning is expected to happen under the current $\hat{M_i}$ while the resource under-provisioning will be forecasted if η_i goes beyond θ (0.98 by default). These two bounds will help the binary search to locate the right $\hat{M_i}$ only if the difference between MLB_{max} and MLB_{min} is larger than UN_{MLB} (set at $(M_{max} - M_{min})/4096$ by default), which indicates the acceptable precision of the

ALGORITHM 1: MLB Exploration Algorithm

Input: The current measure of query latency variation for T_i (i.e., CV_i), the UDRx for T_i (i.e., $u_i(x)$), the UDR target for T_i (i.e., $\hat{U_i}$).
Output: The suboptimal mean latency budget ($\hat{M_i}$).

```
MLB_max ← M_max; MLB_min ← M_min;
repeat
    M̂_i = (MLB_min + MLB_max)/2;
    /*Obtain the UDR U_i under the current M̂_i*/
    U_i ← 0;
    Obtain F_{X_i}(x) according to M̂_i and CV_i based on Eq. 4 and Eq. 5;
    /*Accumulate the UDR contributed by different query latencies.*/
    for j ← 1 to m_i do
        U_i ← U_i + (F_{X_i}(x_j) − F_{X_i}(x_{j−1})) * u_i(x_j);
    end
    /*Resource over-provisioning could happen for T_i.*/
    if U_i/Û_i < ζ then
        MLB_min ← M̂_i;
    end
    /*Resource under-provisioning could happen for T_i.*/
    else if U_i/Û_i > θ then
        MLB_max ← M̂_i;
    end
    /*The preferred MLB.*/
    else
        break;
    end
until (MLB_max − MLB_min ≤ UN_MLB);
return M̂_i;
```

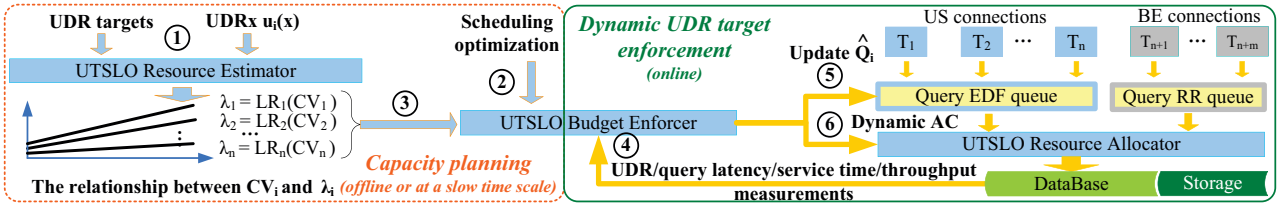

Figure 4: The illustration of UTSLO architecture and workflow.

MLB exploration. The time complexity of Algorithm 1 is $O(log(\frac{M_{max}-M_{min}}{UN_{MLB}}))$, indicating that only a few iterations (12 at the most by default) are required for a MLB exploration.

Discussion: UTSLO captures the entire query latency CDF per tenant's connection for the UDR computation during its MLB exploration. Although the lower latency region has less impact on UDR, it is costly to split the latency CDF per tenant into a low-latency region to be discarded and a high-latency region to focus on, because even if all tenants have the same UDR target, which is not likely, their latency CDF can be highly variable, making the cutoff a moving target and introducing extra computational costs and inaccuracy.

3.3 UTSLO Components

UTSLO aims to implement the goal of the objective function (i.e., Eq. 1), i.e., maximizing the mean latency for each US tenant's connection constrained by its UDR target and freeing up more service capacity of the database system to serve other tenants. This is done through *capacity planning* and *dynamic UDR target enforcement* that are jointly realized by three UTSLO components, i.e., the resource estimator, the budget enforcer, and the resource allocator, as shown in Fig. 4.

Capacity planning: It aims to figure out the highest throughput (i.e., when all the US tenants' connections are query intensive.) required for the US tenants according to their UDR targets (i.e., $\hat{\lambda}^{US}_{tot}$) under the optimized low-level latency variation. Thus, the throughput for the BE tenants is determined under the total throughput that can be provided by the underlying database system minus $\hat{\lambda}^{US}_{tot}$.

Dynamic UDR target enforcement: It targets at enforcing the given UDR targets under highly dynamic query latency variation and resource provisioning, and adapting resource allocation to dynamically changed UDR targets within a predictable convergence time.

As the brain of UTSLO, the resource estimator and the budget enforcer work together to do the capacity planning for the given UDR targets. In addition, the budget enforcer lays the foundation for the UDR target enforcement by optimizing scheduling parameters for the resource allocator. In doing so, the resource allocator can strategically enforce admission control and scheduling policies for concurrent tenants' queries to dynamically ensure US tenants' targets in a cost-effective fashion, while providing the maximum available resources for more tenants (e.g., the BE tenants). As illustrated in Fig. 4, the UTSLO workflow comprises 6 steps. The first three are set for the capacity planning (3.3.1) while the rest form a feedback-loop control for the dynamic UDR target enforcement (3.3.2).

3.3.1 The Capacity Planning. UTSLO's capacity planning involves three steps. As shown in Fig. 4, we first set up UTSLO (in **Step**①) by establishing the relationship between the throughput budget and the CV of latency for each US tenant's connection, and then tune the two-dimensional admission control knobs to limit the multi-tenant query concurrency and per-connection query bursts for optimizing the trade-off between high throughput and low-level latency variation (in **Step**②). The capacity planning can be carried out based on the throughput budget estimation for tenants (in **Step**③).

Step①: The inputs to UTSLO, including the UDR target $\hat{U}_i$ of each US tenant's connection T_i, $1 \le i \le n$, and its individual UDRx $u_i(x)$, are first passed to the resource estimator. It can precisely estimate the mean latency/throughput budgets ($\hat{M}_i$ and $\hat{\lambda}_i$) required for ensuring the UDR target under different levels of latency variation (CV_i) by approximating the latency distribution of T_i's queries as a generalized exponential distribution [35] fitting the UDR target ($\hat{U}_i$) (details in Section 3.1 and 3.2). To estimate the mean latency/throughput budget under highly dynamic query latency variation, it is required to repeat the estimation (i.e., MLB exploration algorithm in Section 3.2) under different levels of latency variation, generating a curve between the throughput budget values ($\hat{\lambda}_i$) and the measurements of query latency variation (CV_i) (i.e., $\hat{\lambda}_i = LR_i(CV_i)$), as shown in Fig. 4.

Step②: To estimate the throughput budget for each given UDR target under the database system for a typical number of concurrent tenants (e.g., hundreds), we need to first bound query latency variation (i.e., $CV_i \le \hat{cv}$) with negligible throughput loss. To this end, UTSLO first adopts the AppleS algorithm[45] to obtain an optimized upper bound of query concurrency C. Further, UTSLO optimizes the upper limit (D) for per-connection query bursts by finding an optimal throughput peak with the minimum D under the concurrency bounded by C. In this way, the side effects caused by unregulated multi-tenant query patterns that aggravate query latency variation can be effectively alleviated almost

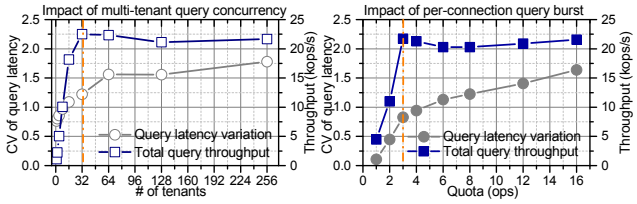

Figure 5: The impact of multi-tenant query concurrency and per-connection query burst on query latency variation and throughput.

without sacrificing throughput. To verify its effectiveness, we first exponentially scale tenants' connections for enterprise-class cloud services and adopt the AppleS algorithm to hide excessive query parallelism beyond the processing capacity of the database system. This helps obtain the peak throughput with the CV of query latency at around 1.22 (as shown in the left subfigure of Fig. 5). And then, based on query concurrency optimization, we further adjust the number of requests allowed for each database connection to issue in a scheduling round, or *quota*, to control per-connection query burst and observe its impact on latency variation and throughput. As a result, combining the optimization efforts of multi-tenant query concurrency and per-connection query burst further reduces the CV of latency to around 0.82 (decrease by 32.8%) at a similar peak throughput, as shown in the right subfigure of Fig. 5, which is a better trade-off between latency variation and throughput.

Step③: Based on Step ① and Step ②, the configurations (i.e., $\hat{\lambda}_i = LR_i(CV_i)$ and $CV_i = \hat{cv}$) will be used to estimate the throughput budget for each US tenant's connection and obtain the total throughput budget for the US tenants ($\lambda_{tot}^{\hat{US}}$). Moreover, we can also know the total throughput provided by the database system (Ψ) from Step ②. Thus, if the throughput required for the US tenants $\lambda_{tot}^{\hat{US}}$ is smaller than its upper limit, UTSLO will consider the UDR targets feasible.

Note that step② is only needed to run once before serving tenants while step① and ③ are only required for newly added tenants, which can run offline or at a slow time scale.

3.3.2 Dynamic UDR Target Enforcement. The capacity planning only offers the static configurations for UTSLO under the given query latency variation and throughput provisioning. Our goal is to dynamically track and regulate multi-tenant latency distributions to accurately meet their individual UDR targets (by **Step**④⑤⑥ in Fig. 4). These three steps form a feedback-loop control to enforce each US tenant connection's target under highly dynamic query latency variation and throughput. Specifically, step ④ aims to collect different measurements from the database system, including the UDR U_i, the query service time S_i, mean latency, latency variation CV_i, and etc., which are statistics over the sliding window (containing $1K$ completed queries by default). Based on the relationship between the throughput budget for the database connection T_i and its query latency variation (i.e., $\hat{\lambda}_i = LR_i(CV_i)$), the throughput budget ($\hat{\lambda}_i$) can be dynamically adjusted under the current measurement of query latency variation (CV_i) and thus the corresponding mean latency budget ($\hat{M}_i$) can be obtained by $\frac{1}{\hat{\lambda}_i}$ according to Little's law [47]. And then, in step ⑤, the queuing time budget ($\hat{Q}_i$) can be derived by $\hat{Q}_i = \hat{M}_i - S_i$, which indicates the maximum time a T_i's query can stay in the queue. The earliest-deadline-first (EDF) scheduler sorts the US tenants' connections in the ascending order of $\hat{Q}_i$. In addition, step ⑥ aims to dynamically coordinate the resource allocation across connections to fully exploit the instantaneous surplus throughput by the two-dimensional admission control (AC) (details in Section 4.2) driven by UDR targets.

In the event of UDR target change or temporary resource under-provisioning, the corresponding mean-latency/throughput budgets are required to be dynamically adjusted to address the observed UDR target violations, which needs a period of *convergence time* to transition. It is necessary to figure out the length of convergence time according to the maximum allowed throughput for the US tenants to offer a predictable convergence time.

UDR needs to collect the disengagement ratio contributed by all the queries that are completed in the reference period S (i.e., S ranging from t_1 to t_2 with the length of $w^S = t_2 - t_1$). Hence, for any US tenant's connection T_i, we can calculate its UDR during S based on UDRx of $u_i(x)$ that is defined on m_i ($m_i > 1$) latency intervals, i.e., $(l_0, l_1]$, $(l_1, l_2]$, ..., $(l_{k-1}, l_k]$, ..., $(l_{m_i-1}, l_{m_i}]$, and $l_0 = 0$, as $U_i^S = \sum_{k=1}^{m_i} \frac{N_i^S(X_i \le l_k) - N_i^S(X_i \le l_{k-1})}{N_i^S} * u_i(l_k)$, where $N_i^S(\le l_k)$ denotes the number of queries with smaller latency than l_k while N_i^S represents the number of queries completed in S.

Thus, if the current UDR exceeds the target $\hat{U}_i$, the length of convergence time can be obtained by $\frac{(U_i^{S_u} - \hat{U}_i) * Q_i}{(\hat{U}_i - U_i^{S_o}) * \lambda_i^{S_o}}$, where $U_i^{S_u}$ and Q_i denote the UDR and the accumulated number of completed queries respectively when UDR target violations happen, while $U_i^{S_o}$ and $\lambda_i^{S_o}$ represent the UDR and the throughput obtained over the convergence time respectively. To shorten the convergence time, UTSLO will minimize the throughput for the BE tenants and offer the maximum throughput it can allocate for the US tenants. Note that $U_i^{S_u}$, Q_i, and $\lambda_i^{S_o}$ are measurable or predictable by the throughput-budget based estimation (see Section 3.1 and 3.2), and UTSLO can reserve adequate processing capacity to respond to UDR target violations to reduce the UDR during the convergence time ($U_i^{S_o}$) to a small value (close to 0). Thus, UTSLO can estimate the convergence time for each US tenant's connection that experiences UDR target violations, for which the effectiveness has been demonstrated in Fig. 10.

4 UTSLO IMPLEMENTATION

Accurate UDR target enforcement under multi-tenant databases highly relies on an easy-to-deploy, effective, and efficient implementation of the UTSLO resource allocator. To this end, it adopts a non-intrusive low-cost query control/scheduling mechanism (Section 4.1). Further, a concurrency-based sharing (CS) is adopted for an efficient fine-grained resource allocation with a low-level query latency variation (Section 4.2). In addition, more efforts are made on exploiting instantaneous surplus database processing capacity by dynamic UDR-target-driven admission control (Section 4.3).

4.1 Non-Intrusive Query Control

Database systems (e.g., MongoDB [11]) receive tenants' queries and send results back to tenants after business-logic-induced query processing, by establishing TCP connections. Thus, by strategically intercepting and suspending network-I/O system calls, the UTSLO resource allocator can effectively realize admission control/scheduling for multi-tenants' connection-level query streams. It is implemented as a user-space module based on a syscall_intercept library [3], to intercept and schedule network-I/O system calls issued by the database system. Thus, UTSLO supports agile deployment and fast upgrade without the need to instrument the database system or OS kernels. The difference among different database systems only lies in the adoption of different types of network system calls to handle queries, where the IP address and port for each connection is resolved. The overhead of the system-call interception is found to be negligibly small, i.e., on the order of 100 nanoseconds, which is sufficient in support of high-performance databases or soft-realtime applications [26].

4.2 Admission Control & Scheduling

The UTSLO resource allocator is designed to realize the efficient multi-tenant connection-level query processing resource allocation to precisely enforce the given UDR targets, which relies on the collaboration between the two-dimensional admission control/scheduling mechanisms. In the view of the resource allocator, the running time of concurrent tenants' connections consists of multiple scheduling rounds, to each of which the admission control and scheduling policies are applied to realize the following control/scheduling goals: 1) Low-level query concurrency and burst. 2) Maintaining high total throughput. 3) Dynamic fine-grained throughput allocation over tenants' connections.

As shown in Fig. 5, a low UDR prefers low-level query concurrency and burst, which are controlled by tuning the two knobs, (1) the multi-tenant concurrency limit P and (2) per-connection query quota ($L_{US[i]}$, $1 \le i \le n$, for n US tenants' connections and $L_{BE[j]}$, $1 \le j \le m$, for BE tenants)

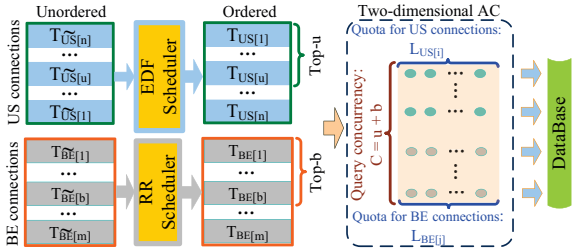

Figure 6: UTSLO admission control/scheduling mechanisms.

respectively, while maintaining as high a total throughput as possible without crossing their individual thresholds (i.e., C and D are the upper bounds for P and per-connection query quota respectively). These thresholds can help to form the boundaries of a feasible two-dimensional region for query admission control (AC), within which these control knobs can be dynamically optimized for the fine-grained resource allocation among tenants' connections to accurately meet the UDR targets with a minimized throughput allocation while providing as high a throughput allocation as possible for the BE tenants.

Since the queuing time budget can set the priority for query throughput allocation across the US tenants' connections (see Section 3.3.2), how to share the total query throughput between the US tenants and the BE tenants in a fine-grained fashion with a low-level query concurrency and burst is challenging. To address the issue, we start by exploring the alternative solutions, i.e., per-connection query *Q*uota based throughput *S*haring (*QS*) or *C*oncurrency based throughput *S*haring (*CS*). QS can roughly enforce the proportion of throughput between the US tenants and the BE tenants by keeping the approximate ratio of query quotas assigned to the US tenants' connections and the BE tenants' connections based on weighted fair queuing, which is widely adopted by storage I/O resource management systems [32, 33, 39, 58]. However, it can lead to serious query bursts that can increase UDR, especially for stringent UDR targets that require high query quotas for the US tenants' connections and thus likely exacerbate their latency variation (as shown in Fig. 5).

In contrast, *CS* enforces the resource sharing by assigning the slots of query concurrency to the top-C tenants' connections, which are chosen by two schedulers, i.e., the EDF and round-robin (RR) schedulers for the US tenants and the BE tenants respectively, on a specific probability of chance for the US tenants (i.e., α). Specifically, the top-C tenants' connections (i.e., u US tenants' connections and b BE tenants' connections, $C = u + b$) will be selected from the tenant list sorted by different scheduling policies and $\alpha = \frac{u}{C}$. The EDF scheduler sorts the US tenants' connections according to the urgency of their latest queuing time budget $\hat{Q}_i$ while the RR scheduler is used for the BE tenants to fairly share query

throughput. In this way, the ratio of query throughput for the US tenants (i.e., p) can be coordinated by adjusting α in a continuous fashion without aggravating latency variation.

4.3 UDR Targets VS. High Throughput

To exploit instantaneous resource over-provisioning, the percentage of throughput allocated to the US tenants (p) and the per-connection quotas need to dynamically adapt to UDR measurements.

Specifically, we adopt a feedback control similar to the low-cost additive-increase/multiplicative-decrease (AIMD) control [4] to adjust α for the p adaption and work with per-connection quota coordination under the constraints of U^{max} and U^{min}. If U^{max} is less than θ (set at 0.98 by default) and U^{min} is less than ζ (set at 0.95 by default), surplus resources are considered available for the US tenants while UDR target violations are deemed imminent if U^{max} is larger than θ. For the former case, α will be updated by $\alpha = \alpha - \frac{\gamma}{\overline{cv}}$ to increase the throughput share for the BE tenants, where γ is set at 0.05 by default and $\overline{cv}$ denotes the average CV of query latency across US tenants' connections, which is used to offset the side effects of unexpected spikes of latency variation on UDR as p decreases. Moreover, the BE tenants' connections are allowed to increase their quotas (limited by D) to further share the surplus throughput while the US tenants' connections will decrease their quotas to reduce the unnecessary risk of query bursts. For the latter case, α will be updated by $\alpha = (1 + \beta) * \alpha$ to prevent the imminent UDR target violations by increasing p (β is set at 0.25 by default). Meanwhile, the BE tenants will back off by minimizing their quotas while each US tenant connection with the UDR (U_i) larger than θ will increase its quota by 1 until it reaches D.

Note that the p adjustment coordinates the throughput share between the BE and US groups. In the mean while, per-connection quota coordination further enables throughput reallocation across tenants' connections within each group, aiming to maximize the throughput share for the BE group without violating the UDR targets under highly variable resource provisioning.

5 PERFORMANCE EVALUATION

Test Environment: All the evaluation experiments are conducted on a dedicated rack of PowerEdge R630 servers. The storage server is equipped with a RAID-0 SSD array with five 800GB SATA MLC Solid State Drives, consolidating all the logical volumes for databases. The computing server is configured with 2 Intel Xeon E5-2650 processors, 64GB of RAM, a Broadcom NetXtreme II BCM57810 10Gb NIC and 4 × 1TB SATA HDDs. All the servers are connected by a Dell N4032F switch with peak bandwidth of 10Gb.

Workloads and Databases: We deploy a key-value store, MongoDB 3.6.0 [11], as a representative NoSQL database to verify the effectiveness of UTSLO on enforcing UDR targets in a multi-tenant environment for enterprise-class cloud services. Specifically, we run Yahoo Cloud Serving Benchmark (YCSB) [25] on MongoDB with multi-tenant connections, each generating a Zipf distributed key-value query workload. Multi-tenant query workloads include different combinations of GET and SET, and are write-heavy (50% GET, 50% SET) unless otherwise noted, accessing the underlying MongoDB that stores a 150GB dataset to simulate intensive key-value query processing generated during cloud services. In addition, to effectively assess the capability of the UDR target enforcement for multi-tenant e-commerce services built on a database system, we adopt Py-TPCC [40] to construct a transactional database consisting of 1,000 warehouses (for a total dataset size of 78GB) on MongoDB 4.4.3 to simulate intensive users' online e-commerce transaction processing through multi-tenants' database connections.

5.1 Baselines and Methodology

UTSLO aims to enforce UDR targets for multi-tenant database systems under different online service scenarios (e.g., enterprise-class cloud services or e-commerce services) in a cost-effective manner. A US tenant can own multiple database connections each of which can have its own UDR target.

We adopt a state-of-the-art any-percentile tail latency SLO guarantee solution, i.e., PSLO [46], as a baseline to verify the effectiveness of UTSLO on UDR target enforcement in terms of its accuracy[5], cost (i.e., the throughput for the US tenants), and the throughput for the BE tenants (a higher value is better)[6]. Since tail latency SLO guarantee solutions can not directly support UDR target enforcement, we use three empirical rules (i.e., *OPT*, *PES*, and *MOD*) to translate each UDR target into its corresponding tail latency SLO. *OPT* assumes that the UDR contributed by the query latencies falling into the unguaranteed percentile range are only higher than the guaranteed percentile range by the smallest margins (e.g., the next step above the guaranteed UDR step in the UDR-latency staircase function of Fig. 2(d)), which is an optimistic estimation and requires the least extra effort/resources. In contrast, *PES*, as a pessimistic estimation and requiring the most extra effort/resources, means that the UDR contributed by the query latencies falling into the unguaranteed percentile range assume the highest UDR (i.e., 100%). The moderate estimation, i.e., *MOD*, aims to estimate

[5]It is evaluated by the percentage error between the actual UDR and the target (a positive value means a UDR target violation and a smaller absolute value indicates a higher enforcement accuracy), or the normalized UDR that is defined by the actual UDR divided by the target (a value larger than 1 means a UDR target violation and a value closer to 1 indicates a higher enforcement accuracy.)

[6]The throughput for the US (or BE) tenants is defined by the number of completed queries for the US (or BE) tenants during the reference period of time divided by its length.

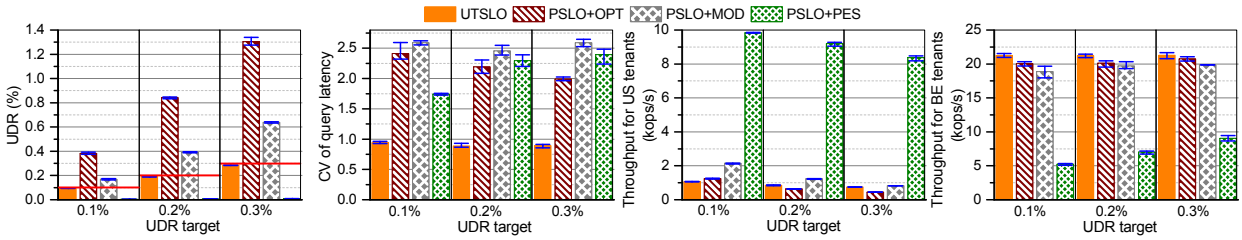

Figure 7: Comparisons in UDR, CV of query latency, and the throughput for the US tenants and BE tenants under different schemes for enterprise-class cloud services.

between the above two extreme cases by averaging their percentiles.

Enterprise-class cloud services built on key-value query processing typically generate much higher query throughput than that under e-commerce services that have to offer ACID guarantees for the correctness of online concurrent transaction processing and compromise throughput performance. Thus, enterprise-class cloud services are more sensitive to query concurrency and burst control and throughput under-provisioning. In contrast, e-commerce services' workloads are characterized by a higher level of query latency variability than enterprise-class cloud services and thus bring a bigger challenge for UTSLO in dynamically tracking and regulating multi-tenant query latency distributions to accurately guarantee UDR targets at the connection-level. Therefore, besides comparing against tail latency SLO guarantee solutions, we'll focus on evaluating the cost-effectiveness of UTSLO in enforcing UDR targets for enterprise-class cloud services under highly variable resource provisioning and dynamic UDR target enforcement. In addition, we'll assess the ability of UTSLO to differentiate different levels of UDR targets accurately for e-commerce services.

Specifically, for the first three subsections, we conduct the experiments for enterprise-class cloud services with up to 512 database connections, where each US tenant has one connection with the UDR-latency function (UDRx) that reflects the rule of thumb reported by Amazon (i.e., each $100ms$ increase in service latency means a 6% drop in sales [15]).

In Section 5.2, we aim to verify the effectiveness and accuracy of UTSLO in UDR target enforcement and assess the ability of UTSLO to track and regulate dynamic multi-tenant query latency distribution under highly dynamic latency variation and throughput. In Section 5.3, we examine the efficiency of different implementations of the UTSLO resource allocator under distinct schemes, i.e., quota-based sharing (QS) and the concurrency-based sharing (CS). In Section 5.4, we verify the effectiveness and robustness of dynamic UDR target enforcement when some US tenants dynamically change their UDR targets. We also verify how closely the UTSLO implementation can approximate its theoretical goals, by obtaining theoretical metrics (e.g., the throughput for the US tenants) based on the model (described in Section 3.1) as another baseline under an ideal case where all US tenants achieve their UDR targets without error, and the query throughput and latency variation are constant. The experiments for e-commerce services in Section 5.5 are conducted with a total of 256 connections. A US e-commerce tenant establishes three connections for the users via three types of devices (i.e., desktop, mobile, and tablet), each of which has its unique UDRx, as shown in Fig. 2(d).

For both online service scenarios, the BE tenants share the rest of database connections and make full use of the residual database query processing capacity. To verify the effectiveness of UTSLO on the UDR target enforcement under high load where online services often suffer from compromised QoS, all the following experiments are conducted under a very high I/O utilization (typically over 95%).

5.2 The Effectiveness and Accuracy of UDR Target Enforcement

In this subsection, we adopt PSLO [46] as the baseline system to enforce the tail latency SLOs obtained under three levels of empirical estimation (i.e., *OPT*, *PES*, and *MOD*), denoted by $PSLO+OPT$, $PSLO+PES$, and $PSLO+MOD$, respectively. To effectively simulate the scenario where the US tenants suffer from a strong I/O interference from many BE tenants, we let 512 tenants, of which only 10 are US tenants, access the underlying database. Thus, the ratio of BE and US tenants is over 50 : 1. For the US tenants, we set their UDR target at 0.1%, 0.2%, and 0.3%, respectively.

As shown in Fig. 7, the high-percentile unguaranteed range left by tail latency SLO enforcement solutions can significantly impact UDR and likely violate the target, especially for *OPT*. Under $PSLO + OPT$, the tail latency SLOs for the three UDR targets (i.e., 0.1%, 0.2%, and 0.3%) are $< P90 : 25ms >$[7], $< P80 : 25ms >$, and $< P70 : 25ms >$, respectively. In fact, PSLO can precisely enforce these tail latency SLOs of $25ms$ for the US tenants in the ranges of $(23.84ms, 24.98ms)$, $(23.97ms, 24.97ms)$, and $(23.72ms, 24.98ms)$ at the $P90$, $P80$, and $P70$, respectively. However, the actual UDR values are 3.75×, 4.11×, and 4.21× their individual UDR targets of 0.1%, 0.2%, and 0.3%, resulting in unacceptable UDR target violations. In contrast, $PSLO + PES$ goes to the other

[7] <Px:y> denotes the tail latency SLO of y at the x^{th} percentile.

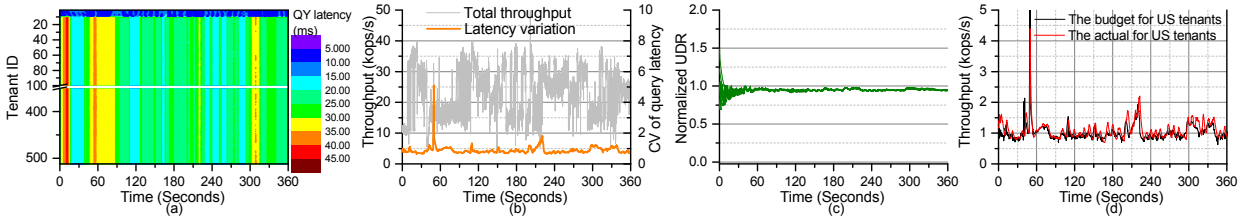

Figure 8: The verification of the capability of UTSLO to enforce UDR targets under highly variable throughput provisioning and query latency variation for enterprise-class cloud services.

extreme, i.e., an over-conservative estimation for the unguaranteed percentile range, resulting in a serious resource over-provisioning. For *PSLO + PES*, the tail latency SLOs for the three UDR targets are $< P99.9 : 25ms >$, $< P99.8 : 25ms >$, and $< P99.7 : 25ms >$, respectively. PSLO can also accurately guarantee these tail latency SLOs of $25ms$ for the US tenants in the ranges of $(22.48ms, 23.60ms)$, $(22.37ms, 24.05ms)$, and $(23.18ms, 24.20ms)$ at the $P99.9$, $P99.8$, and $P99.7$, respectively. Nevertheless, all the UDR values under *PSLO + PES* are close to 0. Although the UDR targets are guaranteed, the throughput for the US tenants required for the UDR targets of 0.1%, 0.2%, and 0.3% are 9.25×, 10.87×, and 11.19× higher than those obtained under UTSLO. As for *PSLO + MOD*, all the UDR targets are violated although the resource over-provisioning for the US tenants is alleviated compared with *PSLO + PES*. Note that the throughput for the US tenants under *PSLO + MOD* for all the three UDR targets is higher than those obtained under *UTSLO*, especially for the targets of 0.1% and 0.2%. This means that UTSLO requires less query throughput than *PSLO* to enforce the UDR target largely because the design/implementation of UTSLO strengthens its ability on cutting down the level of query latency variation and thus reduce the cost of UDR target enforcement. As shown in Fig. 7, the CV of latency variation for the US tenants under *UTSLO* for the three UDR targets are 2.56×, 2.47×, and 2.24× lower than those obtained under *PSLO + OPT*. A similar advantage of UTSLO on reducing the level of query latency variation can also be observed when comparing against *PSLO + PES* and *PSLO + MOD*. All in all, UTSLO provides the highest throughput for the BE tenants among all the schemes with accurate UDR target enforcement (the average percentage error is −5.88%).

Now we further assess the sensitivity of the UDR target enforcement under UTSLO to highly variable query latency variation and throughput when all the UDR targets are 0.1%. We draw two observations from results shown in Fig. 8. First, *temporary throughput under-provisioning does not affect the UDR target enforcement.* Fig. 8(b) shows a U-shaped throughput curve during the time between 39*s* and 85*s*, indicating a period of throughput under-provisioning during which the BE tenants (with the tenant ID ranging from 11 to 512) yield query throughput to the US tenants (the top 10 tenants) and result in a higher query latency for BE tenants (the yellow part), as illustrated by Fig. 8(a). In contrast, all US tenants' UDR values closely approach their individual targets (ranging from 0.92 to 1.00 based on the measurement every $100ms$) but without violating any target, as shown in Fig. 8(c). This means that the throughput diverted from the BE tenants to the US tenants is necessary for the latter's UDR target enforcement. Second, *the throughput for the US tenants can effectively adapt to the fluctuation of query latency variation*, as reflected by the comparison between their variation trends in Fig. 8(b) and (d), respectively. Since the throughput required for the UDR target enforcement is positively associated with query latency variation, UTSLO is designed to adapt the throughput budget for each US tenant to its current level of latency variation to minimize throughput over-provisioning. The resource allocator can precisely allocate the throughput to the US tenants according to their throughput budgets (in Fig. 8(d)), which contributes to accurate UDR target enforcement (in Fig. 8(c)).

5.3 Efficiency of the Resource Allocator

In this subsection, we let 512 tenants, with different ratios of US and BE tenants, access the MongoDB database. Specifically, we exponentially increase the number of the US tenants from 25 to 200 and divide the US tenants into 5 groups with each having a different UDR target, i.e., 0.1%, 0.15%, 0.2%, 0.25%, 0.3%, respectively. In this way, we can assess the effectiveness and efficiency of the UTSLO resource allocator under different implementations (i.e., QS and CS) in terms of the throughput for the US tenants, the throughput for the BE tenants, query latency variation, and normalized UDR, where the minimum, the maximum, and the average are reported for the former three metrics over three runs while the minimum, the maximum, and the average normalized UDR across the US tenants are used to assess the sensitivity of UDR target enforcement to different UDR targets.

For all the cases shown in Fig. 9, the CS implementation provides a smaller throughput for the US tenants than the QS implementation while the throughput allocated to the BE tenants by CS is higher than that by QS. This is because CS can keep a lower level of query latency variation than QS. QS relies on adjusting per-connection query quota to realize resource allocation between the US and BE tenants, likely aggravating query latency variation. More importantly, CS can

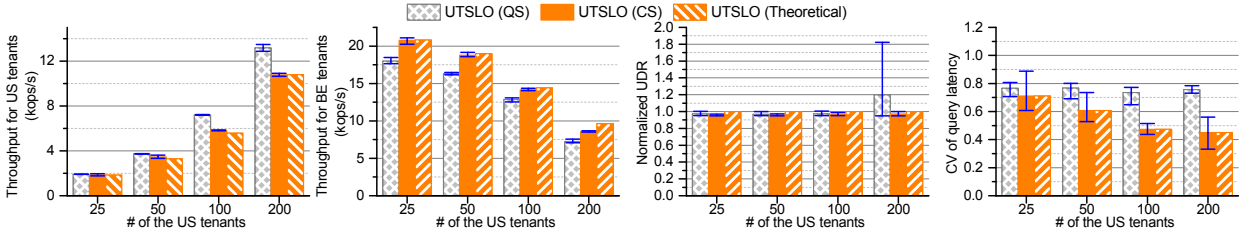

Figure 9: Comparisons in the throughput for the US tenants and the BE tenants, normalized UDR, and CV of query latency under different schemes for enterprise-class cloud services.

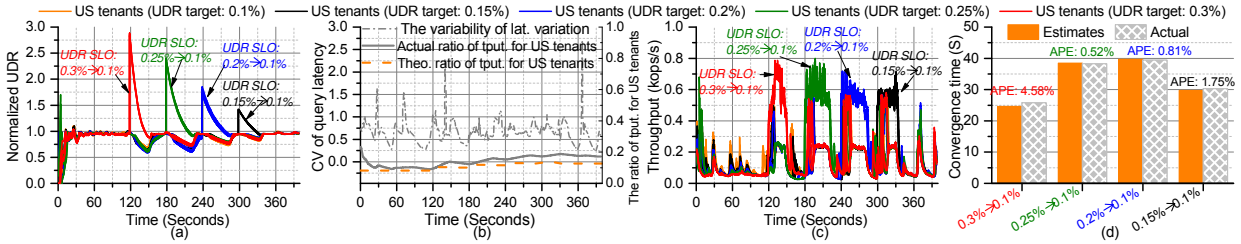

Figure 10: The illustration of dynamic UDR target enforcement consisting of 4 UDR target changes (i.e., 0.3%→0.1%, 0.25%→0.1%, 0.2%→0.1%, and 0.15%→0.1%), each of which involves 5 US tenants while the total number of tenants is 512 for enterprise-class cloud services.

accurately ensure different UDR targets when the number of US tenants increases with the throughput allocated to the US tenants approaching the theoretical values. This means that UTSLO can effectively and accurately track and regulate highly variable multi-tenant query latency distribution required by the given targets, which approximates the ideal situation where the query throughput and latency variation are constant. In contrast, QS shows the fastest increasing rate for the throughput allocated to the US tenants largely due to its quota-base resource allocation mechanism. For the case of 200 US tenants, the CV of query latency under QS is 1.68× higher than that under CS, respectively. Note that CS provides much stabler allocation than QS, for which we must use the best-case QS results from the multiple runs in the figure. Despite this, only the US tenant group with the loosest UDR target, i.e., 0.3%, can meet its target and the US tenant group with the tightest target has a normalized UDR of 1.8 or more, resulting in serious UDR target violations.

5.4 Dynamic UDR Target Enforcement

In this experiment, there are a total of 512 tenants of which 25 are US tenants that are divided into 5 groups of 5 each. We initially assign 5 different UDR targets to the 5 groups, i.e., 0.1%, 0.15%, 0.2%, 0.25%, 0.3%, respectively. At the time of 120 seconds, we first change the UDR target of the group with the target of 0.3% to 0.1%. After that, we change the groups with the targets of 0.25%, 0.2%, and 0.15% of their targets all to 0.1%, one at a time every 60 seconds and in that order. Throughout the process, the maximum allowed throughput share for the US tenants is 25%. And then, we can assess the effectiveness and predictability of dynamic multi-tenant UDR target enforcement under UTSLO.

As shown in Fig. 10, before any UDR target changes, UTSLO can effectively enforce the distinct UDR targets of the 5 US tenant groups accurately (Fig. 10(a)). Fig. 10(b) demonstrates that, from the start of the observation period when the first target is changed, the curve of the actual ratio of the throughput allocated to the US tenants quickly converges to the theoretical one after around 22.5 seconds, where around 8.5 seconds are used to accumulate adequate queries (1024 by default) for each tenant's UDR statistics. After that, we can observe from Fig. 10(a) four similar triangle-shaped spikes each corresponding a UDR target change. When the UDR target of 0.3% is changed to 0.1%, the normalized UDR of the corresponding 5 US tenants steeply increase to near 3.0, before they use about 25.8 seconds to converge to their new UDR target. During the convergence time, as shown in Fig. 10(c), we can find that the instantaneous throughput (obtained every 100*ms*) for the US tenants experiencing UDR target changes significantly increase as UTSLO forces the BE tenants to yield the extra throughput beyond their minimum quota. Accordingly, the actual ratio of the throughput allocated to the US tenants increases from 9.3% to 13.5%. After that, a stable and precise UDR target enforcement is observed until the next round of UDR target changes. Although the actual ratio of the throughput allocated to the US tenants finally decreases to 12.3%, slightly higher than the theoretical one of 10.1%, the actual measure shows an obvious trend to converge to the theoretical curve. We can also observe similar patterns for the subsequent three rounds of UDR target changes. Note that UTSLO can accurately estimate the convergence time (the maximum absolute percentage error (APE) is less than 5%), as shown in Fig. 10(d), which makes dynamic UDR target enforcement predictable.

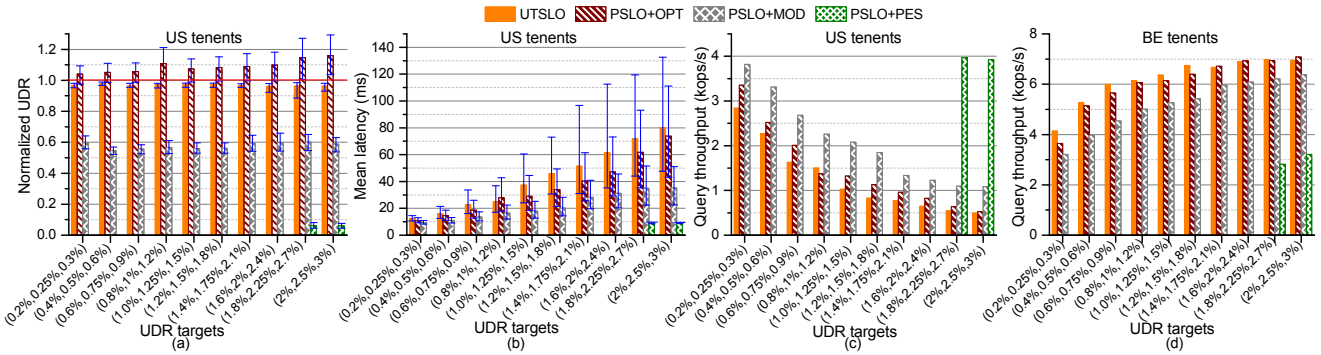

Figure 11: Comparisons in normalized UDR, mean latency, and the throughput for the US tenants and BE tenants under different schemes for e-commerce services.

5.5 E-commerce services

We evaluate the capability of UTSLO to differentiate the UDR targets of three concurrent US tenants, which linearly increases from (0.2%, 0.25%, 0.3%) to (2%, 2.5%, 3%) at increments of 0.2%, 0.25%, 0.3% respectively, and enforcing them accurately for e-commerce services. We also conduct the same experiments under the tail latency SLO solution PSLO with three empirical rules (i.e., OPT, PES, and MOD). The maximum ratio of query throughput for the US tenants is 60% beyond which the scheme is considered as infeasible.

As shown in Fig. 11(a), all the 30 different UDR targets for the 9 US tenants' connections (each US tenant assigns the same UDR target for its three connections at a time) can be accurately met under UTSLO. Specifically, the average normalized UDR over all the cases is 0.969 while the highest normalized UDR is 0.989 (1.0 is the normalized UDR target). In contrast, the schemes of *PSLO + MOD* and *PSLO + PES* conservatively enforce the UDR targets at a high cost. Specifically, as shown in Fig. 11(b), UTSLO only needs 1.85× the mean latency required by *PSLO + MOD* for the same UDR target on average. At the same time, the average normalized UDR obtained under *PSLO + MOD* is only 0.577 over all the cases, which is far below the target. It is no wonder that *PSLO + MOD* requires 79.53% more throughput for the US tenants and 16.87% less throughput for the BE tenants than UTSLO on average for all the UDR targets, as shown in Fig. 11(c)(d). *PSLO + PES*, as the most extreme conservative solution, requires the highest throughput for US tenants over all the schemes and is only feasible for the two loosest targets (i.e., (1.8%, 2.25%, 2.7%) and (2%, 2.5%, 3%)). *PSLO + OPT*, as a solution for UDR target enforcement with optimistic estimation, essentially ignores the impact of high tails happened in the unguaranteed percentile range on the UDR measure and cause much more UDR target violations. As a result, the maximum normalized UDR for *PSLO + OPT* is 28% higher than the target while its average normalized UDR is 1.09 over all the cases. In contrast, UTSLO can dynamically track and regulate the entire query latency distribution for all the US tenants' connections in an efficient fashion by fine-grained resource allocation with a low-level query latency variation. Therefore, *PSLO+OPT* still requires 17.82% more throughput than UTSLO for the US tenants on average for all the UDR target enforcement cases. Note that UTSLO can effectively differentiate and enforce UDR targets highly accurately for e-commerce services.

6 CONCLUSIONS AND FUTURE WORK

We propose UTSLO to cost-effectively bound the latency-induced UDR for multi-tenant database systems. As the brain of UTSLO, the resource estimator and budget enforcer perform capacity planning for precisely guaranteeing UDR targets with a minimized resource over-provisioning. To support the dynamic UDR target enforcement, the UTSLO resource allocator adopts the UDR-target-driven dynamic admission control and scheduling mechanisms to minimize the throughput required for the UDR targets and provides as much throughput as possible for the BE tenants. Our extensive evaluation for both enterprise-class cloud services and e-commerce services demonstrates that UTSLO can accurately enforce per-connection UDR targets (the normalized UDR ranging from 0.941 to 0.999 for all the experimental cases) with minimized throughput provisioning.

For distributed databases based on replica sets or sharded clusters, UTSLO can be applied by guaranteeing UDR targets at the database-instance level that typically involves single replica or data shard. A major challenge is for scatter-and-gather queries that access some or even all the shards (e.g., range queries). In this case, we need to decompose the query UDR target into shard-level ones, which we will consider as our future work.

7 ACKNOWLEDGMENTS

This work is supported in part by the US National Science Foundation grants CNS-2008835 and CCF-2226117. We thank the anonymous reviewers for their valuable comments. We are especially grateful to our shepherd, Alexander Böhm, for helping improve the paper's presentation.

REFERENCES

[1] 2016. Why Brands Are Fighting Over Milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds/?sh=56aacfa04ad3.

[2] 2017. The State of Online Retail Performance. https://s3.amazonaws.com/sofist-marketing/State+of+Online+Retail+Performance+Spring+2017+-+Akamai+and+SOASTA+2017.pdf.

[3] 2019. The system call intercepting library . https://github.com/pmem/syscall_intercept.

[4] 2021. Additive increase/multiplicative decrease. https://en.wikipedia.org/wiki/Additive_increase/multiplicative_decrease.

[5] 2021. Amazon study: Every 100ms in Added Page Load Time Cost 1% in Revenue. https://www.contentkingapp.com/academy/page-speed-resources/faq/amazon-page-speed-study/.

[6] 2022. Bounce rate. https://en.wikipedia.org/wiki/Bounce_rate.

[7] 2022. Churn rate. https://en.wikipedia.org/wiki/Churn_rate.

[8] 2022. Coefficient of variation. https://en.wikipedia.org/wiki/Coefficient_of_variation.

[9] 2022. Conversion rate. https://en.wikipedia.org/wiki/Conversion_marketing#Conversion_rate.

[10] 2022. Db2 workload manager. https://www.ibm.com/docs/en/db2/11.1?topic=administration-db2-workload-manager.

[11] 2022. MongoDB. http://www.mongodb.org/.

[12] 2022. MySQL. http://www.mysql.com.

[13] 2022. Oracle. https://www.oracle.com.

[14] 2022. TRANSACTION PROCESSING PERFORMANCE COUNCIL. The TPC-C home page . http://www.tpc.org/tpcc/.

[15] 2022. Website Speed Matters More Than You Think. https://callchrishenry.com/website-speed/.

[16] 2023. Build a Multi-Tenant Architecture. https://www.mongodb.com/docs/atlas/build-multi-tenant-arch/.

[17] 2023. cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[18] 2023. Managing Resources with Oracle Database Resource Manager. https://docs.oracle.com/en/database/oracle/oracle-database/21/admin/managing-resources-with-oracle-database-resource-manager.html.

[19] 2023. MongoDB Atlas. https://www.mongodb.com/cloud/atlas.

[20] 2023. MSSQL Server: Resource Governor. https://learn.microsoft.com/en-us/sql/relational-databases/resource-governor/resource-governor?view=sql-server-ver16.

[21] 2023. PostgreSQL. https://www.postgresql.org/.

[22] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM*.

[23] Prince Bharadwaj, Sameer Kumar, Ashwin Raj, and Manoj Hudnurkar. 2022. Role of Database Management in E-Commerce Firms. *CHANGING FACE OF E-COMMERCE IN ASIA* (2022), 297–313.

[24] Binlei Cai, Rongqi Zhang, Laiping Zhao, and Keqiu Li. 2018. Less provisioning: A fine-grained resource scaling engine for long-running services with tail latency guarantees. In *In Proceedings of International Conference on Parallel Processing (ICPP)*.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[26] S. S. Craciunas, C. M. Kirsch, and H. Röck. 2008. I/O Resource Management Through System Call Scheduling. *SIGOPS Oper. Syst. Rev* 42, 5 (2008), 44–54.

[27] J. Dean and L. A. Barroso. 2013. The Tail At Scale. *Commun. ACM* 56 (2013), 74–80.

[28] Aaron J. Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng Yan. 2013. Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. In *In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*.

[29] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. 2015. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT*.

[30] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. 2017. Crystal: Software-Defined Storage for Multi-Tenant Object Stores. In *Proccedings of the conference on File and storage technologies (FAST)*.

[31] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. 2015. Queues don't matter when you can jump them!. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*.

[32] A. Gulati, I. Ahmad, and C. A. Waldspurger. 2009. PARDA: proportional allocation of resources for distributed storage access. In *Proccedings of the conference on File and storage technologies (FAST)*.

[33] A. Gulati, A. Merchant, M. Uysal, and Peter J. Varman. 2009. Efficient and adaptive proportional share I/O scheduling. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[34] A. Gulati, A. Merchant, and P. J. Varman. 2010. mClock: handling throughput variability for hypervisor IO scheduling. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[35] R. D. Gupta and D. Kundu. 1999. Generalized Exponential Distributions. *Australian & New Zealand Journal of Statistics* 41, 2 (1999), 173–188.

[36] Sergio Hernández, Pedro Álvareza, Javier Fabra, and Joaquín Ezpeleta. 2017. Analysis of users' behavior in structured e-commerce websites. *IEEE Access* 5 (2017), 11941–11958.

[37] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. 2013. Speeding up Distributed Request-Response Workflows. In *SIGCOMM*.

[38] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. 2015. Silo: Predictable message latency in the cloud. In *SIGCOMM*.

[39] W. Jin, J. S. Chase, and J. Kaur. 2004. Interposed proportional sharing for a storage service utility. *ACM SIGMETRICS Performance Evaluation Review* 32, 1 (2004), 37–48.

[40] Asya Kamsky. 2019. Adapting TPC-C benchmark to measure performance of multi-document transactions in MongoDB. In *Proceedings of the VLDB Endowment*.

[41] Julian Köllerström. 1974. Heavy traffic theory for queues with several servers. I. *Journal of Applied Probability* 11, 3 (1974), 544–552.

[42] J. F. C Kingman. 1961. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society* 57, 4 (1961), 902–904.

[43] Yipkei Kwok, Patricia J. Teller, and Sarala Arunagiri. 2017. 2TL: A scheduling algorithm for meeting the latency requirements of bursty I/O streams at user-specified percentiles. In *IEEE International Conference on Cloud and Autonomic Computing (ICCAC)*.

[44] Zhenyu Leng, Dejun Jiang, Liuying Ma, and Jin Xiong. 2020. Gecko: Guaranteeing Latency SLO on a Multi-Tenant Distributed Storage System. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.

[45] Ning Li, Hong Jiang, Hao Che, Zhijun Wang, and Minh Q. Nguyen. 2022. Improving scalability of database systems by reshaping user parallel I/O. In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[46] N. Li, H. Jiang, D. Feng, and Z. Shi. 2016. PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage.

In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[47] JDC Little. 1961. NA proof for the queuing formula: L= λW. *Operations research* 9, 3 (1961), 383–387.

[48] C. R. Lumb and R. Golding. 2004. D-SPTF: Decentralized Request Distribution in Brick-based Storage Systems. *SIGOPS Oper. Syst. Rev.* 38, 5 (2004), 37–47.

[49] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. 2015. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*.

[50] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[51] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[52] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. 2010. Stout: An adaptive interface to scalable cloud storage. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[53] Jyotiprasad Medhi. 2002. *Stochastic models in queueing theory*. Elsevier.

[54] Vivek Narasayya and Surajit Chaudhuri. 2021. Cloud data services: Workloads, architectures and multi-tenancy. *Foundations and Trends® in Databases* 10, 1 (2021), 1–107.

[55] M. Nguyen, S. Alesawi, N. Li, H. Che, and H. Jiang. 2018. Forktail: a black-box fork-join tail latency prediction model for user-facing datacenter workloads. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing (HPDC)*.

[56] J. Perrya, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. 2014. Fastpass: A centralized zero-queue datacenter network. In *SIGCOMM*.

[57] Eric Schurman and Jake Brutlag. 2009. Performance Related Changes and their User Impact. In *Velocity-Web Performance and Operations Conf*.

[58] D. Shue, M. J. Freedman, and A. Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[59] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*.

[60] B. Vamanan, J. Hasan, and T. Vijaykumarn. 2012. Deadline-aware datacenter tcp (d2tcp). In *SIGCOMM*.

[61] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. 2013. Low Latency via Redundancy. In *CoNEXT*.

[62] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. 2012. Cake: enabling high-level SLOs on shared storage systems. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[63] Z. Wu, C. Yu, and H. V. Madhyastha. 2015. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*.

[64] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*.

[65] J. Zhang, A. Riska, A. Sivasubramaniam, Q. Wang, and E. Riedel. 2006. Storage performance virtualization via throughput and latency control. *ACM Transactions on Storage (TOS)* 2, 3 (2006), 283–308.

[66] T. Zhu, D. S. Berger, and M. Harchol-Balter. 2016. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[67] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. 2014. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.