# FrugalDB: A Cost-Efficient Multi-Tenant Database System

Yifeng Luo [§1], Junshi Guo [§2], Jiaye Zhu [§3], Zhenjie Zhang [†4], Shuigeng Zhou [§5]

[†] *School of Computer Science, Fudan University, China P.R.*
[1,2,3,5]{luoyf,jsguo14,zhujy14,sgzhou}fudan.edu.cn

[†]*Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore*
[4]zhenjie@adsc.com.sg

***Abstract*—Quality of Service (QoS) is at the core of the vision in Database as a Service (DBaaS), which provides guarantees to the database users on the usability of the database service, even when the underlying database infrastructure is shared by multiple users. Traditional approaches in DBaaS reserve computation resources, e.g. CPU and memory, based on the Service Level Objective (SLO) given by the database users, so that the database engine always possesses sufficient resource to accomplish the expected workload under any circumstance. Such resource reservation schemes inevitably result in poor resource utilization, as the actual workload of the tenants are usually way below their maximal workload expectation described in their SLO.**

**To enhance resource utilization and reduce operational cost, we propose a novel architecture in our prototype multi-tenant database system, *FrugalDB*, which generally consolidates more tenants with performance SLOs on one single database server. FrugalDB accommodates two independent database engines, an in-memory engine for high workloads with tight SLOs, and a disk-based engine for less active workload with loose SLOs. The dual processing logic enables huge computation resource saving, by assigning the workloads from the tenants to the appropriate engine for query processing. Different from traditional multi-tenant database system, the design of FrugalDB put emphasis on the migration cost minimization, which is incurred when moving workloads across the engines. Based on a workload estimation, FrugalDB responds quickly to the workload updates with minimal overhead on database migration. We validate and evaluate FrugalDB with extensive experiments, which proves its prohibitively higher tenant consolidation rate with performance SLO guarantees, fewer performance SLO violations and acceptable response latency.**

## I. Introduction

The Internet has enabled and popularized DBaaS [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], where service providers host database applications on their infrastructures, and deliver them as services to different end users (also called tenants), who would otherwise need to deploy these database applications by themselves. Tenants who have subscribed DBaaS services could query and update their data via web browsers or client programs, so the ownership and operation of database applications are outsourced to DBaaS service providers. DBaaS service providers typically consolidate multiple tenants into the same hardware/software platform so as to reduce operational cost via resource sharing.

When tenants necessitate QoS guarantees and require satisfaction of performance SLOs, service providers usually have to reserve resources for tenants according to their performance SLOs, so that a database server would always have enough processing capacity to handle all tenants' workloads consolidated on this server, under various workload conditions. However, service providers could only achieve moderate resource utilizations in such a resource-reservation tenant consolidation fashion. DBaaS tenants often quantify their resource requirement according to the peaks of their workloads, and they usually tend to over-estimate their performance SLOs to make sure that subscribed resources could completely cover the actual resource needs of their businesses, so the workload pressure generated by a tenant will not be persistently intense enough to catch up with this tenant's performance SLO. What's more, multi-tenancy tends to possess low overall tenant activeness, and there exist few chances that massive tenants would generate their workloads at their full speeds concurrently, so the DBaaS system would witness moderate resource utilizations most of the time.

Carlo Curino et al. [**?**] proposed a buffer pool gauging procedure, included in their tenant consolidation scheme named Kairos for their multi-tenant database platform [**?**], to estimate the working set size of a database workload, so that they could properly accomplish consolidation for multiple database workloads with minimum servers. However, this buffer pool gauging procedure is not applicable for our targeting scenario. The difficulty is twofold: firstly, to precisely quantify a tenant's resource demands requires the tenant's workload to be stable, while it is impossible to accomplish precise quantifications if the tenant's workload changes dynamically over time; secondly, when far more tenants are consolidated into the same database, there is no reliable mechanism for guaranteeing that memory resources are allocated to processing workloads which actually require more resources. As a huge amount of tenants are consolidated into the same database, buffer resources allocated to tenants with high-intensity workloads may be spared for other massive numbers of tenants with low-intensity workloads, and precious memory resources are not persistently employed to buffer data for those tenants with high-intensity workloads but wasted on buffering data for tenants with low-intensity workloads. So our targeting scenario

requires an effective mechanism which could dynamically quarantine tenants with high-intensity workloads from tenants with low-intensity workloads, which could guarantee that processing of high-intensity workloads would not affect by low-intensity workloads, and thus the DBaaS system could get the most out of resources allocated to process high-intensity workloads.

Targeting at application scenarios where thousands of small tenants with low overall tenant activeness and yet with various database schemas should be served with QoS guarantees, we propose a novel multi-tenant database system called FrugalDB to further improve resource utilization and reduce operational cost, where small tenants' database sizes are expected to be at tens of or a few hundred megabytes scales and their workloads usually exhibit obvious unstableness with huge variance. FrugalDB could consolidate more tenants with performance SLOs onto the same database server. While for medium or large tenants whose database sizes grow up to gigabytes with strict QoS requirements and stable workloads, we think it would be much more appropriate to adopt the shared-machine scheme, which encapsulates different tenants' database in different properly configured virtual machines (VMs) running independent databases, and it is out of this paper's discussion scope.

We mainly make the following three contributions in this paper: 1) propose a workload offloading mechanism to handle mixed workloads of tenants with performance SLOs; 2) formulate and solve the workload offloading problem as an optimization problem; 3) implement a prototype of the proposed workload offloading mechanism, and perform extensive experiments to evaluate the efficiency of the mechanism. In the remainder of this paper, we first introduce the system design and implementation details about FrugalDB in Sec. **??**, and formally define the targeting problem and present the algorithms employed to solve the problem in Sec. **??**. We provide experimental results to validate FrugalDB's efficiency in Sec. **??**, and present the related work of FrugalDB in Sec. **??**. Sec. **??** concludes this paper.

## II. Related Work

Database-as-a-service (DBaaS) is becoming increasingly popular as a cost-effective data management model for providing services in multi-tenant cloud environments, and there have been extensive research efforts devoted to improving services for multi-tenancy to support mixed workloads in cloud environments w/o performance guarantees.

A lot of previous work focused on virtual machine (VM) configuration, provisioning and consolidation to avoid resource over-provisioning [**?**], [**?**]. Soror et al. [**?**] proposed a method to automatically configure VMs for different database workloads consolidated on the same hardware, and Soundararajan et al. [**?**] presented a system for determining resource partitions for virtual machines. Shen et al. [**?**] presented CloudScale to online predict resource demand of VM-based applications and automate fine-grained elastic resource scaling for virtualized cloud computing infrastructures, resolving scaling conflicts using VM migration. Some prior work looked at dynamic provisioning of VMs for multi-tier web applications, such as Cecchet et al. [**?**] presented Dolly which explore VM cloning technique to spawn database replicas to address the challenges of provisioning shared-nothing replicated databases in clouds. Nathuji et al. [**?**] presented Q-Clouds as a QoS-aware control framework to dynamically tune resource allocations to mitigate performance interference between applications consolidated on the same server. These VM-based consolidation approaches achieve moderate consolidation and are appropriate only when hard isolation between databases is more important than cost or performance. Schiller et al. [**?**] proposed tenant-aware data management to natively support multi-tenancy in database. Narasayya et al. [**?**], [**?**] presented SQLVM to provide isolated performance for multi-tenant DBaaS by reserving resources specified by tenants within database server process, instead of packing resources in the form of VMs. Curino et al. [**?**] and Lang et al. [**?**] presented mechanisms for consolidating multiple databases into the same server, aiming to minimize the number of servers needed to satisfy performance goals for a large number of workloads. Their work is complementary to ours, as it could be used to optimize resource partitions or assignment of workloads which may be generated by our scheme.

Some prior work [**?**] is devoted to achieving extremely high levels of multi-tenancy, where common tables are shared by tens of thousands of tenants with identical or similar databases schemas, and yet with nearly inactive database workloads. Aulbach et al. [**?**] proposed a schema-mapping multi-tenant technique, called Chunk Folding, where tenants' logical tables are vertically partitioned into chunks and folded into different shared physical multi-tenant tables, which may be joined as needed for query processing. Hui et al. [**?**] presented M-Store to consolidate tuples from different tenants into shared tables, where Bitmap Interpreted Tuple (BIT) and Multi-Separated Index (MSI) are proposed to improve scalability and efficiency of multi-tenant database system. This prior work aims at resolving the key challenge of improving databases scalability when extremely large numbers of tables and/or columns need to be handled, and none takes performance SLOs into consideration. So this work is closely related to ours, yet complementary to our work, as our approach may also run into similar internal database limitations.

Some solutions focus on workload queuing and scheduling [**?**], and admission control [**?**] for multi-tenancy, when performance SLOs are taken into consideration. These approaches follows the resource-reservation fashion, which do not increase real processing capacity of database servers and may decline services for tenants when data serving systems become overwhelmed. Their work is complementary to ours, as we could employ these approaches to implement traffic control when our scheme is not able to satiate tenants' performance SLOs anyway. Our main target is to handle high-intensity workloads in time, so as to eliminate violations of performance QoS guarantees during workload-bursty periods.

## III. System Design

As modern CPUs could provide more and more powerful computing capacity, the disk-based storage subsystem has more and more obviously become the bottleneck for OLTP applications, because of mechanical characteristics of modern magnetic disk devices. There exists a basic observation that computation resources are always under-utilized heavily in a typical OLTP database system, while memory and disk bandwidth are scarce resources and are usually in strong need. So the key point for satiating the performance SLO of an OLTP application is whether the database server could provide enough memory resources to host its working set in memory. If the database server is only configured with very limited memory resources and could not completely host an application's working set in memory, the database would have to frequently visit disk pages, which would make the database system witness deteriorated overall performance with degraded throughput and increased query latency, and the database server would only be able to satiate a relatively low performance SLO necessitated by the application. If memory resources configured on the database server are abundant enough to host the application's working set in memory, the database might be able to process most database operations efficiently without touching underlying disks, and the database only needs to access disks for database log or dirty page flushing operations, so the database server could satiate much higher performance SLOs for the application. So in this paper we basically assume that CPU is not the bottleneck for our DBaaS system, and the bottleneck results from limited amount of available memory and disk bandwidth resources. In this section, we mainly present details about FrugalDB's design and implementation. Firstly we introduce FrugalDB's architecture, and then present the models we employ to describe tenants' workloads, and finally state details about implementation techniques of FrugalDB.

### A. System Architecture

As shown in Fig. **??**, FrugalDB fuses a disk-based database and an in-memory database to process tenants' data serving requests, and other key components include:

1. *Access Controller*: which dispatches data serving requests according to the optimized tenant/workload assignments, and keeps log records of data serving requests for each tenants.

2. *Log Analyzer*: which processes tenants' request logs recorded by *Access Controller*, extracts tenants' workload characteristics and generates a workload description for each tenant. All tenants' workload descriptions will be fed to the *Workload Offload Engine*, together with tenants' performance SLOs.

3. *Performance Monitor*: which monitors system performance by tracking corresponding performance metrics(*e.g.* request throughputs and latencies, etc.), and incurs a workload offloading process when it judges that a workload burst occurs, when monitored performance metrics exceed the predefined thresholds.

4. *Workload Offload Engine*: which is the core component of FrugalDB used to implement workload offloading while making sure that the whole system could gain the most workload offloading benefit. It consists of three modules: *Offloading Benefit Evaluator*, *Data Offloader* and *Data Reloader*. When offloading workloads, *Offloading Benefit Evaluator* evaluates workload offloading benefits of all active tenants, and compute optimized tenant assignments by combining tenants' workload characteristic descriptions provided by *Log Analyzer*, performance metrics provided by *Performance Monitor* and tenants' SLOs. *Offloading Benefit Evaluator* employs the algorithm, which will be discussed later in Sec.. **??**, to find an optimal assignment of all tenants' workloads. Then *Data Offloader* begins to offload into the in-memory database workloads of corresponding tenants from the disk-based database. When the workload burst fades out, *Data Reloader* begins to reload into the disk-based database updated data from the in-memory database, so that corresponding memory resources can be released for subsequent workload offloading.

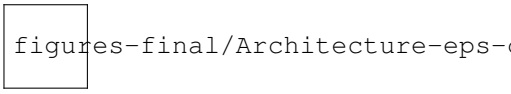



Fig. 1. FrugalDB's architecture.

### B. Workload Models

In a DBaaS multi-tenant environment, a tenant's workload could consist of three different periods: non-active, low-intensity and high-intensity, and various tenant activeness would be presented in different periods. No data serving requests would be yielded during non-active periods, enormous data serving requests would be yielded during high-intensity periods, while a tenant would only yield a moderate number of data serving requests during low-intensity periods, compared with the tenant's performance SLO. If we assume that time consists of sequences of equal intervals, then we could observe that different tenants present various activeness during the same interval, and a tenant's activeness varies across different intervals, which means a tenant's workload may reside in the high-intensity status in an interval, and then turn to the low-intensity status in the subsequent interval, and vice versa. For massive numbers of small tenants with low overall activeness, most tenants' workloads are normally expected to reside in non-active periods or low-intensity periods during most intervals, and only a trivial number of tenants would generate high-intensity workloads during the same interval, which may catch up with their performance SLOs. In this paper, we only take tenants' requirements on query throughput as the evaluation metric when considering QoS.

The basic assumption we hold about tenants' workloads is that *a tenant follows some predictable and repeatable pattern to generate the tenant's workload*. This assumption is reasonable because a tenant usually needs to execute similar business logics at similar time, and thus the overall workload generated by the tenant would statistically follow some pattern

conforming to the tenant's business logics, even if some individual operations of the tenant may seem independent and random with no patterns. Curino et al. [**?**] validated that a tenant's past workload behavior is a good predictor of the tenant's future behavior by experiments with real-life workload datasets, Schaffner et al. [**?**] also revealed that tenants' workloads exhibit obvious load patterns by analyzing realistic tenant workload traces collected in a multi-tenant DBaaS environment, and Elmore et al. [**?**] further characterized tenants' behaviors for tenant placement and crisis mitigation in multi-tenant DBMSs, so we have good reasons to expect that a tenant's workload could be predictable and could be generated repeatedly over time.

Based on this basic assumption, we derive two kinds of models to describe tenants' workloads applicable in our targeting problem: the deterministic model and non-deterministic model, and we believe both of these models could be applicable for specific scenarios respectively. One important thing worth noting is that: it is difficult and unnecessary to model tenants' workloads at a too tiny time granularity in our targeting scenario, for example to build a model to depict a tenant's workload generated at an exact second, as tenants' activities may present huge variance at such a tiny time granularity, so we only need to model tenants' workloads at a coarser time granularity. We thus partition continuous time into discretized time intervals, and then model to predict tenants' workloads during different intervals, by analyzing workload traces collected during past similar intervals. As the workload traces usually are composed of separate queries submitted at different time instants, we compute the average throughput during each interval for each tenant, and then leverage these piecewise query throughput to model tenants' workloads, just as shown in Fig. **??**.

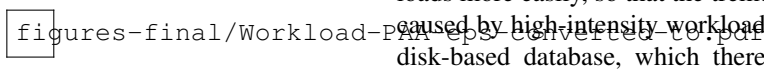



Fig. 2. Workload PAA.

*1) Deterministic Workload Model:* In the deterministic model, we expect that a tenant would generate roughly identical workload pressure to the DBaaS system at similar intervals, that is to say, we assume that a tenant's workload would be generated repeatedly in a deterministic fashion, and we could precisely predict the tenant's workload at time $t$ in advance. So we expect that tenants generate their workloads periodically, and there exists a workload period $T$ for each tenant, where the workload period $T$ could be caliberated by hours, days, weeks, or even months, so that we could assume that a tenant would generate similar workload pressures during interval $t$ and intervals $t + n * T(n = 1, 2, ...)$.

*2) Non-Deterministic Workload Model:* In the non-deterministic model, a tenant's workload would be generated in a non-deterministic fashion, where the workload pressure generated by a tenant during an interval is deemed as a random event. In order to predict a tenant's workload during interval $t$, we model a tenant's workload by statistically computing the distribution of the tenant's historical workload pressures generated at similar past intervals $t - n * T(n = 1, 2, ...)$, where $T$ is deemed as the workload period for defining similar times. As a tenant's workload pressure yielded during an interval is quantified by discrete averaged throughput, we range a tenant's workload pressure into different levels for simplicity consideration, and then model the tenant's workload by statistically computing the distribution of workload pressure levels generated at similar intervals. For example, we could divide a tenant's workload pressures into Low, Middle and High these three levels, and then statistically compute the probabilistic distribution of these three workload pressure levels for each similar interval.

For each tenant, we have to obtain a historical workload trace collected during a long enough duration, so that we could have enough samples to build an accurate probabilistic model for the tenant, otherwise the abstracted probability model would not be able to reflect the actual randomness of a tenant's workload activities. We assume that enough workload traces could be collected by DBaaS service providers and we could employ them to build a realistic probability model.

## IV. Runtime Migration

FrugalDB implements its data serving engine by seamlessly integrating disk-based and in-memory databases, where the disk-based database is responsible for processing massive numbers of tenants' low-intensity workloads, and the in-memory database is mainly responsible for processing a relatively moderate number of tenants' high-intensity workloads. FrugalDB takes advantage of the high processing capacity provided by the in-memory database to temporarily offload high-intensity workloads from the disk-based database, as the in-memory database could handle these high-intensity workloads more easily, so that the tremendous data serving pressure caused by high-intensity workloads could be relieved from the disk-based database, which thereby could focus on serving massive number of tenants with low-intensity workloads. By combining the disk-based database and the in-memory database, FrugalDB could dynamically separate low-intensity workloads from high-intensity workloads and guarantee that tenants with high-intensity workloads could be served with enough crucial resources, so that differentiated services could be provided for tenants with various performance SLOs, whose workload pressures yet change dynamically.

The whole process of offloading a tenant's workload consists of three stages: 1) Data Migration: FrugalDB firstly loads the tenant's data into the in-memory database, and the disk-based database still keeps the original copy of the data; 2) Query Processing: after the tenant's data has been loaded into the in-memory database, all queries submitted by the tenant are directed from the disk-based database to the in-memory database for processing; 3) Data Reverse Migration: when the tenant's workload falls from high-intensity to low-intensity and FrugalDB needs to spare memory resources in the in-memory database to offload other high-intensity workloads, it reversely offloads the tenant's workload from the in-memory database to

the disk-based database, where updated data in the in-memory database is synchronized into the disk-based database, and then all queries submitted by the tenant are directed to the disk-based database for processing.

*3) Query Processing:* When the overall workload pressure imposed on the DBaaS system is moderate and the disk-based database suffices to handle all active tenants' workloads, it will process all queries submitted by all tenants. When more and more tenants' workload pressure turn into high-intensity and the disk-based database is not be able to serve all active tenants, some tenants' high-intensity workloads will be offloaded into the in-memory database, and it will process all queries submitted by these tenants whose workloads are offloaded, while queries submitted by other tenants will still be processed by the disk-based database.

For a tenant whose workload should be offloaded into the in-memory database, the disk-based database continues to process the tenant's queries until all data of the tenant has been loaded into the in-memory database, and the in-memory database is ready to takes over query processes. As data loading takes time, queries which may change a tenant's data status could be submitted during the process of data loading, and thus data which has been loaded into the in-memory database is not consistent with data which still resides in the disk-based database. So we have to handle a tenant's queries properly during workload offloading, especially for queries which would change the tenant's database status, which include insert, update and delete, as these three kinds of queries will cause data inconsistency problem for the tenant, since both the disk-based database and the in-memory database separately possess the tenant's data copies.

We add four column attributes: *rowId*, *isInserted*, *inUpdated*, *isDeleted*, to each tenant's table both in the disk-based database and in the in-memory database, where the rowId attribute of a table is used to locate a row in the table quickly, which increases by 1 each time a row is inserted into the table, and isInserted, inUpdated and isDeleted these three attributes are employed to identify the type of update operation performed on a row. When a tenant's database status changes during workload offloading, FrugalDB keeps record of the update opreation by properly setting these three attributes for corresponding rows: all of these three attributes of each row is set to be "FALSE" originally; when a row is inserted, its rowId attribute gets set properly and its isInserted attribute is set to "TRUE"; when a row is updated, its isUpdated attribute is set to "TRUE", and if a row is deleted, its isDeleted attribute is set to "TRUE".

After the in-memory database begins to handle processing of a tenant's workload, any further operations that change the tenant's database status should also be recorded in the in-memory database: when a row is inserted, its rowId attribute gets set properly and its isInserted attribute is set to "TRUE"; when a row is updated, its isUpdated attribute is set to "TRUE", and if a row is deleted, its isDeleted attribute is set to "TRUE".

*4) Data Migration and Reverse Migration:* When FrugalDB loads a tenant's data into memory tables in the in-memory database, these memory tables' isInserted, inUpdated, isDeleted attributes are all set to "FALSE". After a tenant's data has been loaded into the in-memory database, FrugalDB collects all rows which have been changed since the time when FrugalDB begins to load the tenant's data into the in-memory database, and then replay all operations recorded in these collected rows in the in-memory database without changing any of isInserted, inUpdated, isDeleted attributes of corresponding memory tables: inserts rows whose isInserted attributes are "TRUE" into the corresponding memory tables, updates rows whose isUpdated attributes are "TRUE" with the latest data version, and delete rows whose isDeleted attributes are "TRUE". As replaying these operations on memory tables is very efficient, we can block further operations which will change the tenant's database status temporarily, and execute these newly issued operations on the memory tables after replaying of all previous operations finishes and the in-memory database begins to take over processing the tenant's workload.

When FrugalDB needs to reversely offload a tenant's workload into the disk-based database, any operations which have change the tenant's database status should get reflected into the disk-based database, which will update corresponding tables in the disk-based database properly, and update operations that should be performed on these tables are illustrated in Table. **??**, which are dependent on the status combinations of isInserted, inUpdated, isDeleted attributes of each row contained in the memory tables. For rows whose attribute combinations result in "INSERT" operations, INSERT commands should be called to insert these rows into tables in the disk-based database; for rows whose attribute combinations result in "UPDATE" operations, UPDATE commands should be called to UPDATE these rows; for rows whose attribute combinations result in "DELETE" operations, DELETE commands should be called to delete these rows from the MySQL tables; and rows whose attribute combinations result in "IGNORE" operations will be ignored directly.

As there still exist a few chances that memory tables may still get updated during the process of writing back into the disk-based database, after having collected rows which should be written back to the disk-based database, FrugalDB first deletes all rows which are tagged "isDeleted" from memory tables, and then sets all of these three isInserted, inUpdated, isDeleted attributes to "FALSE" for all remaining rows in these memory tables. When performing updates to the tenant's database in the disk-based database, new updates happen to memory tables, FrugalDB processes these update operations following normal procedures. After all previous update operations are reflected into the disk-based database, FrugalDB recollects those newly updated rows and replays these update to tables in the disk-based database one more time. As it is assumed that the tenant's workload has fallen to low-intensity, we expect that there are only a very limited number of new updates to the memory tables, and the second data writeback should be finished very soon. So any further operations sub-

mitted by the tenant should be blocks temporarily until all updates have been reflected into the disk-based database, after which subsequent queries submitted by the tenant are directed to the disk-based database for execution.

TABLE I
UPDATE OPERATIONS TO THE DISK-BASED DATABASE.

| Attribute Value | | | Update Operation |
|---|---|---|---|
| isInserted | isUpdated | isDeleted | |
| FALSE | FALSE | FALSE | IGNORE |
| TRUE | FALSE | FALSE | INSERT |
| FALSE | TRUE | FALSE | UPDATE |
| FALSE | FALSE | TRUE | DELETE |
| TRUE | TRUE | FALSE | INSERT |
| TRUE | FALSE | TRUE | IGNORE |
| FALSE | TRUE | TRUE | DELETE |
| TRUE | TRUE | TRUE | IGNORE |

## V. MIGRATION SCHEDULER

### A. Problem Formulation

Here we formally present definitions to notations about tenants and database servers as follows. The set of all tenants is denoted as $T$, and each tenant $T_i \in T$ is described as a quadruple $\{S_i, C_i, D_i, W_{i,t}\}$, where $S_i$ denotes $T_i$'s performance SLO (namely the maximum workload that may be generated by $T_i$), $C_i$ denotes $T_i$'s workload characteristics, $D_i$ denotes $T_i$'s data size and $W_{i,t}$ denotes $T_i$'s workload at time $t$. For simplicity, we quantify tenants' SLOs by three levels: High Level, Middle Level and Low Level, and we take the percentage of write requests over total requests as the only workload characteristic metric, which identifies a tenant's workload as Write-Heavy (WH) or Read-Heavy (RH), forming the two possible values of $C_i$. Furthermore, all active tenants at time $t$ are denoted as active tenants $T_{a,t}$. We consider the size of $T_{a,t}$ as a constant proportion over the size of all tenants. We denote the average percentage of write requests over all requests submitted to the data serving engine at time $t$ as $\overline{P_t}$. We denote the maximum request processing capacity of both disk-based database and in-memory database at time $t$ as $W_{M,t}$ and $W_{V,t}$ respectively, and deem them as functions of $\overline{P_t}$, where $W_{M,t} = F(\overline{P_t})$ and $W_{V,t} = G(\overline{P_t})$. $M_V$ denotes the maximum size of memory configured for data storage in the in-memory database. Finally, we define *workload bursts* as circumstances when the overall workload submitted to the data serving engine exceeds the maximum request processing capacity of the disk-based database.

We then formalize the crux of our solution as an optimization problem. When a workload burst occurs to the data serving engine at time $t$, our goal is to assign each active tenant to one of the three tenant sets: $T_{M,t}$, $T_{Vo,t}$ and $T_{Vi,t}$, so as to minimize violations of QoS guarantees for meeting tenants' performance SLOs, where performance SLOs of tenants in $T_{M,t}$ will be met by requesting data from the disk-based database, performance SLOs of tenants in $T_{Vo,t}$ will be met by requesting data from the in-memory database, and performance SLOs of tenants in $T_{Vi,t}$ will be omitted and these tenants' data serving requests should be rejected, as the data serving engine does not possess enough capacity to meet all tenants' performance SLOs at time $t$. So our optimization goal is to minimize:

$$|T_{Vi,t}|, \tag{1}$$

subject to:

$$T_{M,t} \cup T_{Vo,t} \cup T_{Vi,t} = T_{a,t} \tag{2}$$

$$T_{M,t} \cap T_{Vo,t} = \emptyset, T_{M,t} \cap T_{Vi,t} = \emptyset, T_{Vo,t} \cap T_{Vi,t} = \emptyset \tag{3}$$

$$\sum_{m \in T_{M,t}} W_{m,t} \leq B_M * W_{M,t} \tag{4}$$

$$\sum_{v \in T_{Vo,t}} D_v \leq M_V \tag{5}$$

$$\sum_{v \in T_{Vo,t}} C_v * D_v + \sum_{v \in T_{Vo,t}} W_{v,t} \leq W_{V,t}. \tag{6}$$

The first two constraints guarantee that each active tenant is exactly assigned to one tenant set of $T_{M,t}$, $T_{Vo,t}$ and $T_{Vi,t}$. The third constraint guarantees that the overall workloads of active tenants assigned to the disk-based database does not exceed its maximum request processing capacity. The fourth constraint guarantees that the overall data size of active tenants assigned to the in-memory database does not exceed its memory size. The fifth constraint guarantees that the combined workload pressure resulted from workload offloading plus normal data serving requests submitted by tenants assigned to the in-memory database does not exceed its maximum request processing capacity.

### B. Optimization Algorithms

Finally, we present the algorithms proposed to solve our targeting optimization problem. We first introduce algorithms employed to optimize our targeting scenario where tenants' workloads are generated according to the deterministic model, while another targeting scenario where tenants' workloads are generated according to the non-deterministic model could be optimized in similar fashion, only that we employ the expectations of corresponding probability parameters which are used to describe each tenant's workload.

Before a workload burst occurs, all tenants's data serving requests are handled in the disk-based database. As it takes time to generate and finish executing a workload offloading plan, FrugualDB needs to look ahead at future time ranges to judge whether a workload burst is about to occur, so that those highly active tenant's high-intensity workloads could be dealt with by the in-memory database, and the huge workload processing pressure yielded by these high-intensity workloads could be really relieved from disk-based database, and it could focus on processing those low-intensity workloads. If FrugalDB otherwise does not begin to migrate data of high-intensity workloads into the in-memory database ahead of time, it may not be able to successfully offload these high-intensity workloads from the disk-based database in time, as these workloads may have fallen from high-intensity to low-intensity when FrugalDB finishes data migration.

We subsequently assume that a tenant's workload in time $t$ is predictable, and FrugalDB can start the workload offloading ahead of time $t$ at time $t'$, which means $t' < t$, so we can ignore the first part in Constraint. **??**, and get a new constraint:

$$\sum_{v \in T_{Vo,t}} W_{v,t} \leq W_{V,t}. \tag{7}$$

However, this constraint can also be ignored because of the high performance of the in-memory database, where we deem the in-memory database' bottleneck is its memory capacity, rather than its query processing capacity. Thus we finally totally remove Constraint. **??** and just consider Constraint. **??**, Constraint. **??**, Constraint. **??**, Constraint. **??**. Note that the total workload at time $t$ is fixed, and we denote it as:

$$W_t = \sum_{m \in T_{M,t}} W_{m,t} + \sum_{v \in T_{Vo,t}} W_{v,t} + \sum_{u \in T_{Vi,t}} W_{u,t}. \tag{8}$$

So our main goal is to maximize the second part of Equation. **??** subjected to constraints other than Constraint. **??**. If this goal is achieved, we can conclude that the value:

$$\sum_{m \in T_{M,t}} W_{m,t} + \sum_{u \in T_{Vi,t}} W_{u,t} = \sum_{x \notin T_{Vo,t}} W_{x,t} \tag{9}$$

is minimized. If this value satisfy the following condition:

$$\sum_{x \notin T_{Vo,t}} W_{x,t} \leq B_M * W_{M,t}, \tag{10}$$

we can just assign all the tenants who is in the set $T_{a,t}$ but not in set $T_{Vo,t}$ to set $T_{M,t}$, and get the best expected result: $|T_{Vi,t}| = 0$.

However, we may not be able to satisfy Condition. **??** often than not, therefore we must decide which tenants should be assigned to set $T_{Vi,t}$ subjected to Constraint. **??**, so that the object function could be minimized. For this problem, we can just sort the tenants according to their workload from high to low, and assign tenants from top to bottom to set $T_{Vi,t}$ until Constraint. **??** is to be violated. Then the remaining and most important part is how to maximize the second part of the Equation. **??** subjected to the Constraint. **??**. This problem is a classical 0/1 knapsack problem, and we can adopt a dynamic programming solution to solve the 0/1 knapsack problem. We could define a function $F(i, m)$ as the maximum of second part of the Equation. **??**, when we consider the first $i$ tenants in the set $T_{a,t}$, subjected to the following constraint:

$$\sum_{v \in T_{Vo,t}} D_v \leq m. \tag{11}$$

Note that Constraint. **??** is different from Constraint. **??**, where $M_v$ is replaced by the value $m$. So $F(n, M_v)$ is the optimization value we need to compute. It is obvious that:

$$F(0, m) = 0, 0 \leq m \leq M_v. \tag{12}$$

For $i > 0$, we have:

$$F(i,m) = \begin{cases} min(F(i-1,m), F(i-1,m-D_i) + W_{i,t}), \\ \qquad\qquad\qquad\qquad D_i \leq m \leq M_v \\ F(i-1,m), 0 \leq m < D_i \end{cases} \tag{13}$$

We also define a function $P(i, m)$ as:

$$P(i,m) = \left\{ \begin{array}{ll} -1, & i = 0 \\ 0, & (i > 0) \wedge (m < D_i \vee F(i,m) \neq \\ & \qquad F(i-1, m-D_i) + W_{i,t}) \\ 1, & i > 0 \wedge m \geq D_i \wedge F(i,m) = \\ & \qquad F(i-1, m-D_i) + W_{i,t} \end{array} \right\} \tag{14}$$

We use function $P$ to find out which tenants should be assign to set $T_{Vo,t}$, and Alg. **??** and Alg. **??** describe tenant assignment process and the overall algorithm for solving 0/1 knapsack problem respectively. In Alg. **??**, L1 and L2 initialize two sets of tenants $T_{M,t}$ and $T_{Vi,t}$, L3 calls Alg. **??** to compute the value F(n,$M_v$) and the set $T_{Vo,t}$, L4 to L10 put all the tenants in set $T_{M,t}$ except those who are already in set $T_{Vo,t}$, and calculate the total workload produced by these tenants. L11 to L16 decide which tenants will finally be assigned to set $T_{Vi,t}$. In Alg. **??**, L1 to L5 initialize the set of tenants who will be moved to in-memory database, the function F and the function P, L6 to L20 solve the 0/1 knapsack program through dynamic programming, L21 to L29 compute the set of tenants who are going to be moved to the in-memory database.

**Algorithm 1** Tenant assignment

**Require:** $t$, $T_{a,t}$,$M_v$,$W_{M,t}$
1: $T_{M,t} \leftarrow \{\}$
2: $T_{Vi,t} \leftarrow \{\}$
3: Run Algorithm 2 to get $F(n, M_v)$ and $T_{Vo,t}$
4: $W_t \leftarrow 0$
5: **for** $i = 1$ to $n$ **do**
6: **if** i is not in set $T_{Vo,t}$ **then**
7: $W_t \leftarrow W_t + W_{i,t}$
8: $T_{M,t} \leftarrow T_{M,t} + \{i\}$
9: **end if**
10: **end for**
11: **while** $W_t > W_{M,t}$ **do**
12: $id \leftarrow$top($T_{M,t}$)
13: $W_t \leftarrow W_t - W_{id,t}$
14: $T_{Vi,t} \leftarrow T_{Vi,t} + \{id\}$
15: $T_{M,t} \leftarrow T_{M,t} - \{id\}$
16: **end while**
**Ensure:** $T_{M,t}, T_{Vo,t}, T_{Vi,t}$

## VI. Experimental Evaluation

In this section, we validate FrugalDB's feasibility and demonstrate the efficiency of FrugalDB with extensive experiments. We show that not only much higher tenant consolidation but also lower query processing latency could be achieved on FrugalDB provides, when tenants' performance SLOs should be satiated, compared with various baseline experimental settings, which include multi-tenant database implementations purely based on the memory-based database and the disk-based database.

**Algorithm 2** Maximize workload in the in-memory database

**Require:** $t$, $T_{a,t}$,$M_v$
$T_{Vo,t} \leftarrow \{\}$
**for** $m = 0$ to $M_v$ **do**
  $F(0,m) \leftarrow 0$
  $P(0,m) \leftarrow -1$
**end for**
**for** $i = 1$ to $n$ **do**
  **for** $m = 0$ to $M_v$ **do**
    **if** $m \geq D_i$ **then**
      $F(i,m) \leftarrow min(F(i-1,m), F(i-1,m-D_i) + W_{i,t})$
      **if** $F(i,m) = F(i-1,m-D_i) + W_{i,t}$ **then**
        $P(i,m) \leftarrow 1$
      **else**
        $P(i,m) \leftarrow 0$
      **end if**
    **else**
      $F(i,m) \leftarrow F(i-1,m)$
      $P(i,m) \leftarrow 0$
    **end if**
  **end for**
**end for**
$id \leftarrow n$
$capacity \leftarrow M_v$
**while** $id \neq 0$ **do**
  **if** $P(id, capacity) = 1$ **then**
    $T_{Vo,t} \leftarrow T_{Vo,t} + \{id\}$
    $capacity \leftarrow capacity - D_i d$
  **end if**
  $id \leftarrow id - 1$
**end while**
**Ensure:** $F(n, M_v), T_{Vo,t}$

### A. Experimental Setting

All experiments are conducted on two physical machines, which are interconnected with a 1000M high-speed Ethernet switch. One of the two machines runs FrugalDB as the server machine to process tenants' workloads, the other runs threads as clients to simulate tenants' activities. The server machine is equipped with eight Intel(R) Core(TM) i7-4770K CPU and 24GB memory, installed with Ubuntu Linux, MySQL(version 5.5) and VoltDB(enterprise edition 4.9), and the client machine is equipped with six eight-core Intel Xeon 1.87GHz CPUs and 16GB memory, also installed with Ubuntu Linux.

As there are few real-life multi-tenant OLTP workloads readily available, we generate synthetic workloads based on TPC-C and YCSB [**?**] to benchmark the performance of FrugalDB and corresponding baseline settings. TPC-C and YCSB are popular standard database benchmark specifications, consisting of read-only and update operations that simulate the activities of complex database applications. As FrugalDB's targeting application scenarios mainly concern tenants with relatively simple data access requests, we extract all read and update operations contained in TPC-C and YCSB queries, and then leverage these operations to generate the benchmark workloads, where each tenant request only contains one of these extracted operations. Our aggregate benchmark workloads are composed of tenants with corresponding performance SLOs, different data sizes and different read/write percentages, as Table. **??** shows. The value of a tenant's performance SLO specifies the upper bound of requests that the tenant will generate within one minute, for example a tenant whose performance SLO is 100 Requests/Min could at most send 100 requests to and get corresponding responses for these requests from the multi-tenant platform. For the consideration of simplicity, we set three levels of performance SLO: Low, Medium and High, where tenants of the same level are set with the same performance SLO. In this paper, we benchmarked our systems with three Low-Medium-High performance SLO combinations: 5-50-500, 20-60-200 and 100-150-200, where the 5-50-500 combination is leveraged as our default setting, which represents scenarios where tenants require diverse performance SLOs. The 20-60-200 combination represents scenarios where tenants require less diverse performance SLOs, while the 100-150-200 combination represents scenarios where tenants require almost equal performance SLOs. Benchmarking with these three performance SLO combinations could comprehensively reveal FrugalDB's applicability under various scenarios.

For each experiment setting, we run an experiment five times to measure the performance of FrugalDB and other baseline implementations, and we report the average of corresponding metrics as the final results. Each experiment lasts for forty minutes, consisting of eight five-minute time intervals. Note that not all tenants would be concurrently active at the same time in real-world multi-tenant environments, we thus leverage tenant active ratio in our settings to represent the percentage of active tenants over all tenants, and we set the default tenant active ratio to 30%. As a tenant's activeness changes over time, active tenants across different time intervals vary. So we randomly choose a part of active/inactive tenants and let them become inactive/active, maintaining the overall number of active tenants unchanged across various intervals. During each interval, we first follow Poisson distributions to assign each active tenant with a workload intensity, whose expected value amounts to the tenant's performance SLO, and then maintain the tenant's workload stable at the assigned intensity throughout the whole interval. If the assigned workload intensity exceeds a tenant's performance SLO, the tenant's workload intensity is assigned to the tenant's performance SLO.

As we mainly want to evaluate FrugalDB's performance of handling workload bursts, we set the third, the fourth and the seventh intervals out of the eight five-minute intervals to be workload-bursty intervals, and during these bursty intervals all active tenants' expected workloads are set to their performance SLOs, which are aggregately adequate to outstrip the processing capacity of the disk-based MySQL database. Tenants' expected workloads during other non-workload-bursty intervals are only set proportionately to their performance

SLOs, and the proportion is set to 20% in our settings. FrugalDB gains chances during non-workload-bursty intervals to migrate tenant data into/from the memory-based VoltDB database from/into the MySQL database for the purpose of workload offloading. Finally, the memory storage capacity of the memory-based VoltDB database is an important factor which impacts FrugalDB's performance, as it determines the number of tenants whose workloads could be offloading from the MySQL database into the VoltDB database, and we set FrugalDB's default memory storage capacity to 2000MB.

TABLE II
WORKLOAD SUMMARY.

| Feature | Setting | | |
|---|---|---|---|
| SLO | L(30%) | M(50%) | H(20%) |
| TPC-C Data Size(MB) | 6.9(50%) | 16.3(30%) | 35.2(20%) |
| YCSB Data Size(MB) | 10(50%) | 20(30%) | 29(20%) |
| Write/Read Percentage | W(20%) | R(80%) | — |

### B. Experimental Results and Analysis

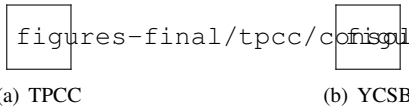
We compared FrugalDB's performance on satisfying tenants' performance SLOs and responsiveness to tenant queries with our baseline settings. Firstly, we define whether a tenant's performance SLO is satisfied or not as: if a tenant is supposed to finish $A$ operations in a minute, while actually sends $B$ queries and gets $C$ query responses in that minute, then we say that the tenant has $A - C$ queries not being fulfilled; if $A - C$ equals zero, then we say this tenant's performance SLO is satisfied during this minute; otherwise we say this tenant's performance SLO is violated and $A - C$ queries are violated for the tenant during that minute. We reported statistics about the number of tenants whose performance SLOs are violated during each minute as the metric for evaluating FrugalDB's performance on satisfying tenants' performance SLOs, and the percentage of query latencies as the metric for evaluating FrugalDB's responsiveness to tenants' queries.

*1) Comparison with Static-SLO:* We compared FrugalDB with the method proposed by Lang et al. [**?**] by simulation, which aims to minimize the number of servers, of different types with different hardware configurations and different processing capacities, needed to satisfy the performance SLOs for a group of tenants. They only considered tenants' SLOs and do not take the dynamic changes of tenants' workloads into consideration, so this method belongs to the static workload consolidation scheme with SLO guarantees, compared with dynamic solutions just as FrugalDB, and we call this method Static-SLO in our paper. As FrugalDB combines disk-based and in-memory these two kinds of databases, we could consider that our multi-tenant platform possesses two different types of databases servers, namely the disk-based database servers and the in-memory database servers, and we implemented the Static-SLO scheme based on these two types of servers. We firstly benchmarked the tenant consolidation that our FrugalDB database server could yield, and then we computed the minimal number of two different types of databases servers that the Static-SLO scheme needs to satiate so many tenants' SLOs, and thus acquired the workload consolidation plan, assigning a tenant's workload to a corresponding server. As cloud service providers usually leverage workload migration strategies to reduce overall operational cost of the whole cloud platform, we combine the Static-SLO scheme with a workload migration strategy to handle situations that FrugalDB targets, where tenants' workloads may change dynamically and dramatically.

The workload-migration-based Static-SLO scheme always tries to satiate all active tenants' SLOs with as few servers as possible by migrating workloads between servers. When it is possible to satiate all active tenants' SLOs with less servers, it powers off servers and migrates workloads running on powered-off servers to remaining servers; when the running servers are not able to satiate all active tenants' SLOs, it initiates servers and then migrates workloads to the newly-initiated servers so that all active tenants' SLOs could be satisfied. We mainly benchmarked the amount of data migration required to satiate all tenants' performance SLOs for FrugalDB and the workload-migration-based Static-SLO scheme, and the results are presented in Fig. **??**. We can see that the amount of data migration required by the workload-migration-based Static-SLO scheme far exceeds that required by FrugalDB, what more important is that data migration of the former happens between servers across network, while data migration of latter only happens within FrugalDB's database server, between its two different databases. So FrugalDB presumably could satiate a group of tenants' SLOs with less servers while with better performance.


Fig. 3. Data migration with dynamic and static SLO satisfaction.

*2) Tenant Consolidation:* Firstly we measured the number of tenants that could be served without performance SLO violations, namely the capacity for tenant consolidation, that different multi-tenant implementations could support for TPC-C and YCSB workloads. Here we compared FrugalDB's performance under the non-deterministic workload model with the implementation purely based on the VoltDB database, and the resource-reservation implementation purely based on the MySQL database. We measured the number of tenants whose workloads amount to their performance SLOs that the MySQL database could support, and take this number as the capacity of tenant consolidation that the MySQL database could provide with resource reservation.



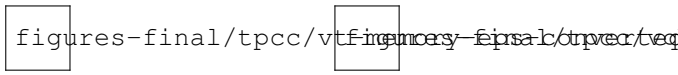

(a) TPCC (b) YCSB

Fig. 4. Tenant consolidation.

Fig. **??** presents the evaluation results of the tenant con-

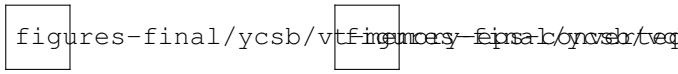
solidation capacity obtained on various multi-tenant database implementations under the non-deterministic workload model, and we can see that: the implementation purely based on the VoltDB database provides the lowest tenant consolidation; the resource-reservation implementation purely based on the MySQL database provides two times higher tenant consolidation than the VoltDB-based implementation; while FrugalDB could achieve 15 times higher tenant consolidation than the VoltDB-based implementation, which is almost 5 times higher than the resource-reservation MySQL-based implementation. It is not difficult to imagine that the VoltDB-based implementation would provide the lowest tenant consolidation, as it needs to load a tenant's data into memory before it could begin to process the tenant's workload, and the memory storage capacity bottlenecks its tenant consolidation capacity. The resource-reservation MySQL-based implementation is bottlenecked by the storage subsystem, where huge amounts of disk accesses are required, and thus intense contention for memory buffer resources is incurred, when the total number of tenants being served increases beyond its processing capacity. FrugalDB could provider much higher tenant consolidation, mainly because of two reasons: workloads for highly active tenants are separated from workloads for tenants with low activeness, where having the high-performance VoltDB database process workloads for most highly active tenants guarantees that enough crucial resources, namely memory buffer resources, are spared for these tenants, and are not impacted by numerous tenants with low activeness, which still reside in the MySQL database to share disk bandwidth, memory buffer and CPU resources; massive numbers of non-active tenants reside in the MySQL database and consume few resources. We did not benchmark the tenant consolidation capacity under the deterministic workload model, which could provide even higher tenant consolidation than the non-deterministic workload model, as the expected high-intensity workloads offloaded into VoltDB under the non-deterministic workload model exist chances of not actually turning out to be high-intensity workloads, while all workload offloading under the deterministic workload model would be effective.

*3) Satisfaction of tenants' SLOs:* Firstly, we benchmarked the satisfaction of tenants' performance SLOs obtained both on the multi-tenant database implementation purely based on the MySQL database and on FrugalDB with various memory storage capacity. As comparison experiments between FrugalDB and the MySQL-based implementation should be performed with the same configurations, the MySQL-based implementation could not follow the resource-reservation fashion, which could only support far less tenants than FrugalDB. So it follows the best-effort fashion without any control mechanism of guarantee tenants' performance SLOs, and Fig. **??** and Fig. **??** present the evaluation results. We can see that the number of tenants whose performance SLOs are violated on MySQL far exceeds on FrugalDB, and the number of tenants whose SLOs are violated on FrugalDB decreases as its memory storage capacity increases, while all tenants' SLOs are satisfied on FrugalDB when its memory storage capacity is set to 2000MB.




(a) SLO-violated Tenants (b) Not-Answered Queries

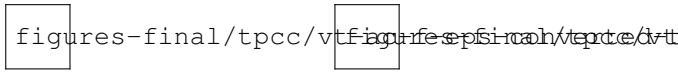
Fig. 5. Satisfaction of tenants' performance SLOs for various implementations, with TPCC.




(a) SLO-violated Tenants (b) Not-Answered Queries

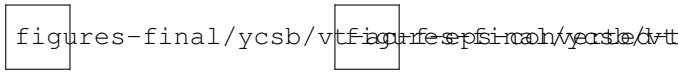
Fig. 6. Satisfaction of tenants' performance SLOs for various implementations, with YCSB.

Secondly, we benchmarked the satisfaction of tenants' performance SLOs obtained on FrugalDB and the MySQL-based implementation with various tenant active ratios, and the evaluation results are presented in Fig. **??** and Fig. **??**. We can see that: on the MySQL-based implementation, the number of tenants whose SLOs are violated increases by 80∼120 as tenants active ratio increases by 5% for TPC-C workloads, by 120∼140 for YCSB workloads; on FrugalDB, all tenants' SLOs are satisfied until tenant active ratio increases to 30%, when a moderate portion of tenants' SLOs could not be guaranteed as the overall number of active tenants exceeds FrugalDB's processing capacity, and the number of tenants whose SLOs are violated increases to around 200 for TPC-C workloads and 380 for YCSB workloads, which indicates FrugalDB's application scenarios.




(a) FrugalDB (b) MySQL-based implementation

Fig. 7. Satisfaction of tenants' performance SLOs with TPCC, with various tenant active ratios.

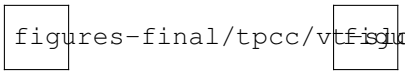



(a) FrugalDB (b) MySQL-based implementation

Fig. 8. Satisfaction of tenants' performance SLOs with YCSB, with various tenant active ratios.

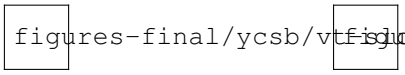
Finally, we benchmarked the satisfaction of tenants' performance SLOs obtained on FrugalDB and the MySQL-based implementation with different Low-Medium-High performance SLO combinations, and the evaluation results are presented in Fig. **??** and Fig. **??** respectively. We can see that: different Low-Medium-High performance SLO combinations engender huge impacts on both FrugalDB's and MySQL's performance, where FrugalDB could guarantee all tenants' SLOs for TPCC workloads while a moderate few tenants' SLOs do not get guaranteed, when the SLO combination is 20-60-200, while a prohibitive number of tenants' SLOs are violated when

the SLO combination is set to 100-150-200 both for TPCC and YCSB workloads, which makes FrugalDB's performance only slightly better than the MySQL-based implementation, this is because tenants' workload intensity contrast is not obvious when the SLO combination is set to 100-150-200, which leaves little optimization space for FrugalDB and the superiority of the workload offloading mechanism does not get embodied apparently. The impact of the SLO combination also indicates FrugalDB's application scenarios, that is to say, FrugalDB could provide better tenant consolidation for multi-tenant database applications where tenants' activeness and workload intensity present sharp divergences.




(a) SLO-violated tenants (b) Not answered queries

Fig. 9. Satisfaction of tenants' SLO with TPCC, with different Low-Medium-High performance SLO combinations.

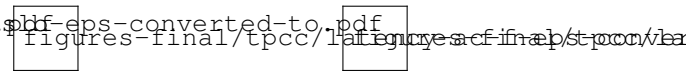



(a) SLO-violated tenants (b) Not answered queries

Fig. 10. Satisfaction of tenants' SLO with YCSB, with different Low-Medium-High performance SLO combinations.

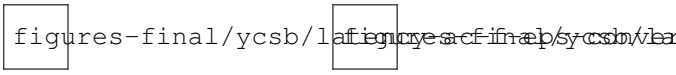
*4) Query latency:* We compared FrugalDB's response latency for tenant queries with the multi-tenant database implementations purely based on the MySQL database and purely based on the VoltDB database respectively, with different experiment settings. The evaluation results obtained on FrugalDB and the MySQL-based implementation with various tenant active ratios are presented in Fig. **??** and Fig. **??**. We can see that: FrugalDB provides overall shorter response time for tenant queries, and tenant active ratio casts little impact on response latency for most tenant queries. This is mainly because FrugalDB needs to execute data loading for workload offloading, which may impact its responses to a part of tenant queries. Fig. **??** and Fig. **??** presents the evaluation results achieved on FrugalDB and on the MySQL-based implementation with various tenant SLO combinations and with different workload models. We can see that: the MySQL-based implementation provides faster responses to most tenant queries, while a minor portion of tenant queries' response latency exceeds FrugalDB, especially when the SLO combination is set to 100-150-200; FrugalDB's responses to most queries with different workload models possess similar latency, while the MySQL-based implementation provides faster responses. Note that experiments with different workload models were performed with the overall tenant number setting to the tenant consolidation of FrugalDB with the deterministic workload model.

Besides, we benchmarked comparatively the response latency for tenant queries obtained on FrugalDB, the VoltDB-based implementation, and on the resource-reservation MySQL-based implementation. Due to page limitation, we only present experimental results with TPCC workloads, and the evaluation results are illustrated in Fig. **??**. Note that these experiments were performed with the overall tenant number setting to the tenant consolidation that the VoltDB-based implementation and the resource-reservation MySQL-based implementation could provide respectively. We can see that: FrugalDB responds faster to tenant queries than both the VoltDB-based implementation and the resource-reservation MySQL-based implementation. It is surprising that the VoltDB-based implementation could only provide slower responses than FrugalDB, this is because most of its memory resources are consumed on storing tenants' data and then it only has fewer memory resources available for query processing, which results in decreased parallelism of query processing in VoltDB. While the workload offloading mechanism implemented by FrugalDB could guarantee that high-intensity workloads are offloaded from the MySQL database and VoltDB possesses enough memory resources to process these high-intensity workloads in the meanwhile.



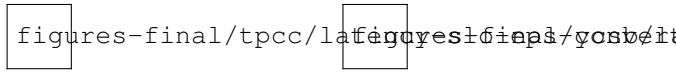

(a) FrugalDB (b) MySQL

Fig. 11. Query latency of FrugalDB and MySQL for TPCC, with various tenant active ratios.



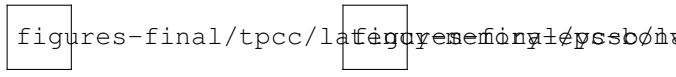

(a) FrugalDB (b) MySQL

Fig. 12. Query latency of FrugalDB and MySQL for YCSB, with various tenant active ratios.




(a) TPCC (b) YCSB

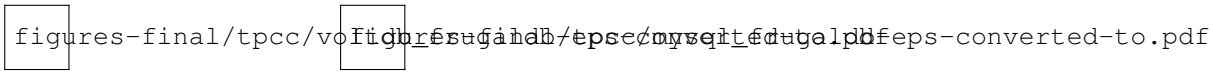
Fig. 13. Query latency of FrugalDB and MySQL, with various tenant SLO combinations.




(a) TPCC (b) YCSB

Fig. 14. Query latency of FrugalDB and MySQL, with various memory storage capacities.

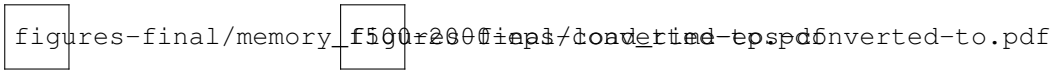
*5) Memory consumption and data migration cost:* At last, we report statistics about memory consumption of the VoltDB database contained in FrugalDB and the time cost of migrating data into/out of the VoltDB database from/into the MySQL database throughout the whole experiments, and Fig. **??** presents the evaluation results. we can see that: the aggregate memory consumption of the VoltDB database increases rapidly




(a) FrugalDB vs VoltDB (b) FrugalDB vs resource-reservation MySQL

Fig. 15. Query latency of VoltDB and resource-reservation MySQL with TPCC.

as its memory storage capacity increases, and increasing the memory storage capacity by 500MB makes the aggregate memory consumption increase 1000MB∼1500MB, because VoltDB needs more memory buffer resources to process increased workloads; the aggregate memory consumption decreases after the workload bursts, because there is few tenants still residing in the VoltDB database, and memory resources consumed could be released to the operating system; the time of loading tenants' data into VoltDB of course increases linearly as its memory storage capacity increases, and it takes about 150 seconds to load 500MB tenants' data, and this is the reason why we should initiate the workload offloading process in advance; while migrating updated data from VoltDB into MySQL for data persistence only takes 1/4 of the time spent on data loading, as only a part of the whole data is updated and needs to be persisted in the MySQL database.




(a) (b)

Fig. 16. (a)Memory consumption, (b)Cost of data migration.

## VII. Conclusion

To further improve tenant consolidation and reduce operational cost, we propose a novel workload offloading mechanism to implement DBaaS with performance guarantees. We aim at scenarios where only a moderate portion out of massively numerous tenants would be simultaneously active to generate enough requests to catch up with their performance SLOs. This mechanism employs a disk-based database to process massive data serving requests from tenants with low-activeness, and an in-memory database to temporarily offload high-intensity workloads from the disk-based database, when it does not suffice to guarantee all active tenants' performance SLOs. We build a cost model to comprehensively evaluate the offloading benefits of active tenants' workloads, and choose those workloads which could contribute most to minimize violations of performance guarantees. We validate and evaluate our scheme with extensive experiments. Experimental results show that our scheme could handle workload bursts efficiently and achieve impressively high tenant consolidation. To the best of our knowledge, our work is the first ever trying to serve massive number of tenants by employing in-memory database technique for multi-tenancy.