SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service

Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, Surajit Chaudhuri
Microsoft Research
Redmond, WA, USA
{viveknar, sudiptod, manojsy, badrishc, surajitc}@microsoft.com

ABSTRACT

A relational Database-as-a-Service provider, such as Microsoft SQL Azure, can share resources of a single database server among multiple tenants. This *multi-tenancy* enables cost reduction for the cloud service provider which it can pass on as savings to the tenants. However, resource sharing can adversely affect a tenant's performance due to resource demands of other tenants' workloads. Service providers today do not provide any assurances to a tenant in terms of isolating its performance from other co-located tenants. We present *SQLVM*, an abstraction for performance isolation which is built on a promise of *reservation* of key database server resources, such as CPU, I/O and memory, for each tenant. The key challenge is in supporting this abstraction within a DBMS without statically allocating resources to tenants, while ensuring low overheads and scaling to large numbers of tenants. Our contributions are in (1) formalizing the above abstraction of SQLVM; (2) designing mechanisms to support the promised resources; and (3) proposing low-overhead techniques to objectively meter resource allocation to establish accountability. We implemented a prototype of SQLVM in Microsoft SQL Azure and our experiments demonstrate that SQLVM results in significantly improved performance isolation from other tenants when compared to the state-of-the-art.

1. INTRODUCTION

Services, such as Microsoft SQL Azure, which offer relational Database-as-a-Service (*DaaS*) functionality in the cloud, are designed to be *multi-tenant*; a single database server process hosts databases of different tenants. Figure 1 illustrates such a multi-tenant RDBMS architecture, called *shared process multi-tenancy*. Multi-tenancy is crucial for cost-effectiveness since dedicating a machine for each tenant makes the service prohibitively expensive. Such multi-tenancy in DaaS is also relevant for on-premise clouds where a single server consolidates databases of multiple independent applications within the enterprise.

An important consequence of multi-tenancy is that a tenant's workload competes with queries from *other* tenants for *key resources* such as CPU, I/O, and memory at the database server. Tenants of a relational DaaS platform can execute arbitrary SQL queries that can be complex and whose resource requirements can be substantial and widely varied. As a result, the performance of a tenant's workload can vary significantly depending on the workload issued concurrently by other tenants. Such performance unpredictability arising from contention with other tenants for shared database server resources can be a serious problem. Therefore, a natural question to ask is: what *assurances* on performance can a multi-tenant DaaS provider (or system) expose to a tenant and yet be cost-effective?

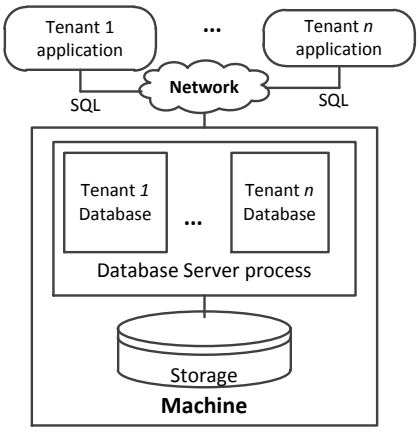

Figure 1. A multi-tenant database system.

It might be tempting to consider assurances of high-level performance metrics at the level of SQL queries, e.g., throughput (queries/sec) or query latency. However, even on a database server that is *exclusively* used by one tenant, the resource needs and execution times of different instances of a single query template, such as a parameterized stored procedure, can vary dramatically depending on parameter values. Moreover, a tenant's workload can have a mix of various types of queries with very different throughput and latency requirements. In addition, observe that service providers need to support ad-hoc queries (i.e., queries not seen previously) without limiting the workload type or the SQL query language supported. Furthermore, a tenant's data size, distribution, and access patterns can change over time. These factors contribute to even greater variability in query throughput and latency. Thus, given the need to support complex and arbitrary SQL workloads, meaningful assurances at the level of queries/sec or query latency, while a worthwhile aspiration, are not even well defined.

A fundamental challenge, however, is to reduce the variability in performance that arises due to contention with other tenants for critical shared database server *resources*. That is, provide an assurance that a tenant's workload is *unaffected* by the workloads executed by co-located tenants. One approach is to provide tenants assurances at the level resources such as CPU, I/O, buffer pool

6th Biennial Conference on Innovative Data Systems Research (*CIDR '13*). January 6-9, 2013, Asilomar, Califonia, USA

memory for caching database pages, and working memory for operators such as hash and sort. At first glance, it may appear that techniques developed for resource management in traditional enterprise DBMS may be adequate for such resource-level assurances. These techniques are typically based on relative priorities, proportional sharing, or enforcing maximum limits on resource usage. However, particularly in a public cloud setting, a major drawback of relative priorities and proportional sharing is that the assurance of how much resources a tenant will receive is not absolute – it depends on which other tenants are active (and their priorities/shares). Similarly, enforcing maximum limits also suffers from this drawback when the service provider *overbooks* (i.e., promise more resources in aggregate to tenants than the available system capacity) to increase utilization and cost-effectiveness [14]. In contrast, a promise of a *reservation* of resources is much more meaningful to a tenant of DaaS, since the assurance of how much resources the tenant will receive is absolute – i.e., not dependent on other tenants. As a concrete example, consider the I/O resource. Suppose tenant T_1 is promised a reservation of 100 I/Os per second (*IOPS*). Then, the promise is that if T_1's workload demands 100 IOPS (or more), then the system assumes responsibility for granting 100 IOPS no matter which other tenants are executing concurrently on the server.

Observe that we do not want to support such resource reservations through static resource allocation since that would drastically limit consolidation and increase the costs. It is therefore possible that a tenant may not always receive the resources it was promised (e.g., due to overbooking). Thus, *metering* the promise becomes crucial to establish accountability, i.e., the system must be auditable so that when the promise is not met, it is possible to determine if this violation occurred because the service provider allocated the resource to other tenants instead, or the tenant's workload had less demand for the resources than it reserved. Referring to the I/O example above, suppose T_1's workload actually achieves 80 IOPS when its reservation is 100 IOPS. There are two reasons why this might happen: (a) T_1's queries did not generate sufficient I/O requests; (b) T_1 generated sufficient I/O requests, but the database system allocated IOPS to other tenants instead, thereby depriving T_1 of some of the resource it was promised. Note that such metering is *independent* of the actual resource allocation mechanisms, and is essential for providing performance assurances in a multi-tenant environment.

In the *SQLVM* project at Microsoft Research, we adopt the above approach to performance isolation. *SQLVM is a reservation of a set of resources for a tenant inside the database server*. Conceptually, the tenant is exposed a familiar abstraction of a virtual machine (VM) with a specified set of resources such as CPU, I/O, and memory, but inside the database server. Internally, new promise-aware resource allocation mechanisms exercise fine-grained control to orchestrate shared resources across tenants without requiring static allocation upfront. If a tenant's resource reservation is not met, then metering logic for that resource establishes accountability. Note that the obvious alternative of actually creating VMs (one per tenant) and running an instance of the database server process within each VM is too heavyweight and fails to achieve the degree of consolidation demanded for DaaS [1]. In contrast, SQLVM is much more lightweight, allowing consolidation factors of hundreds of tenants. Another key advantage of SQLVM is that it applies to *any* RDBMS workload without restrictions.

There are multiple challenges in the design and implementation of the above abstraction within an RDBMS. First, since static resource allocation is not cost-effective, *scheduling mechanisms* in the DBMS need to change with the new constraints of fine-grained resource sharing while meeting each tenant's reservation; each resource brings unique challenges. Second, the implementation of these mechanisms needs to scale to hundreds of active tenants with acceptably low overheads. Last, metering intuitively requires tracking a tenant's demand for resources at a fine granularity while keeping the bookkeeping overheads low.

We briefly touch upon various other important issues that arise when building an end-to-end multi-tenant system using this approach. First, in addition to the tenants' workloads, a database server consumes resources for *system tasks* necessary to achieve other crucial properties such as high availability (e.g., via replication to other machines), checkpointing to reduce recovery time, and backup/ restore. SQLVM also isolates a tenant's resource reservation from such system management activities. Interestingly, such system activity can also be governed via the SQLVM abstraction by logically treating the system as an "internal tenant" with certain resource requirements. Second, service providers often overbook the system to reduce costs by increasing consolidation. Thus, if at any point, the combined resource requirements of all *active* tenants exceed the available resources, the provider will be unable to meet the promises of all tenants. In such a scenario, additional *policies* within the system are necessary to determine which tenants' promises will be violated. These policies may take into account potentially conflicting considerations: fairness to tenants and the need to minimize penalties incurred when the promise is violated. Resource scheduling in the presence of such promises can be viewed as an online optimization problem. Last, while SQLVM adds value by isolating tenants contending for resources, it can also be viewed as a *building block* upon which higher-level performance assurances (e.g., at the workload level) can be designed. For example, SQLVMs of different "sizes" (e.g., *Large*, *Medium*, *Small*) can potentially be exposed, where each size corresponds to a set of reservations of individual resources. This could enable "recommender tools" to profile the tenant's workload against different sized SQLVMs and suggest one that is suitable to meet the tenant's higher-level performance goals.

The contributions of this paper can be summarized as follows:

- An abstraction for performance isolation in a multi-tenant RDBMS based on promise of reservation of resources.
- New fine-grained resource scheduling mechanisms with the goal of meeting each tenant's reservations.
- Novel metering logic to audit the promise.
- An implementation of SQLVM and the associated metering logic inside Microsoft SQL Azure. Our experiments demonstrate that tenants achieve significantly improved resource and performance isolation from other tenant workloads when using SQLVM.

2. SQLVM

SQLVM is a reservation of key resources in a database system such as CPU, I/O, and memory. Conceptually, a tenant is promised a VM with specified resources, but *within* the database server process. Unlike a traditional VM, a SQLVM is much more lightweight since its only goal is to provide resource isolation across tenants. We believe that this abstraction is well suited to a multi-tenant DaaS setting since the assurance provided to a tenant is absolute, i.e., the promise is not specified relative to other active tenants (unlike assurances based on priorities or proportional sharing based on tenant weights). In a cloud setting, a SQLVM can be mapped to a *logical server* (similar to Amazon EC2, for instance), thus making

the promise independent of the actual capacity of the physical server hosting the tenant.

The SQLVM abstraction is also accompanied by an independent *metering* logic that provides accountability to tenants. When a tenant is not allocated resources according to the promise, metering must decide whether the tenant's workload did not have sufficient demand to consume the resources promised or whether the service provider failed to allocate sufficient resources; this logic is unique for each resource promised. A key challenge in metering stems from the burstiness in requests—in the presence of such bursts, metering must be fair to both tenants and the provider. We assume there is a *metering interval*, i.e., a window of time over which this metering is done. For example, if the metering interval is 1 second, then the promised reservation must be met every second.

In the rest of this section, we define the notion of reservation and metering for each key DBMS resource: CPU, I/O, memory (both buffer pool and working memory). In principle, SQLVM can be extended to encompass other shared resources as well, e.g., network bandwidth. Our experiments (Section 4) show that a SQLVM with promises for the resources discussed below already results in much improved performance isolation.

2.1 CPU

Database servers today run on processors with multiple cores. For example, on a machine with two quad-core processors, the server can run up to 8 *tasks* (i.e., threads) concurrently, or more if for example there is hyper-threading. On each core, a scheduler decides which among the tasks queued on that core gets to run next. A task can be in one of the following states: *running* (currently executing on the core), *runnable* (ready to execute but is waiting for its turn), or *blocked* (waiting on some resource, e.g., a lock on a data item, and hence not ready to execute). For a tenant, and a given core, the *CPU utilization* over an interval of time is defined as the percentage of time for which a task of that tenant is running on that core. This definition extends naturally to the case of k cores as the total time for which tasks of that tenant run across all cores, as a percentage of (k × time interval).

Promise: SQLVM promises to reserve for the tenant (T_i) a certain *CPU utilization*, denoted by $ResCPU_i$. This promises T_i a slice of the CPU time on available core(s) and does not require statically allocating an entire core (or multiple cores) for a tenant. This allows better consolidation since we can promise CPU utilization to many more tenants than available cores. For example, on a single core server, if *ResCPU* = 10%, then in a metering interval of 1 sec, the tenant should be allocated CPU time of at least 100 msec, provided the tenant has sufficient work.

Metering: The key challenge in metering CPU utilization is in defining the notion of sufficient work for a tenant in terms of CPU use. We observe that if a tenant has *at least* one task that is running or is runnable, then it has work that can utilize the CPU. Thus, the metering problem can be stated as follows: of the total time during which the tenant had at least one task running or runnable, it must receive at least $ResCPU_i$ percentage of the CPU; the provider violated the promise otherwise. For instance, if T_1 was promised $ResCPU_1$=10% and if T_1 had at least one task ready to run (or running) for 500ms, the provider violates the promise only if the allocated CPU is less than 50ms, i.e., T_1's effective utilization is less than 10%. This definition of metering is fair since the provider is not held accountable for the tenant being idle (i.e., no tasks ready to run), while ensuring that a provider cannot arbitrarily delay a tenant's task without violating the promise.

2.2 I/O

Achieving adequate I/O *throughput* (IOPS) and/or I/O *bandwidth* (bytes/sec) is important for many database workloads. As in the case of CPU, statically dedicating a disk (or set of disks) per tenant to achieve acceptable I/O throughput limits the amount of consolidation. Thus, fine-grained sharing of the IOPS available from a disk is important. For simplicity in the discussion below we refer to I/O throughput, although the definitions can be extended for bandwidth as well. Note that the maximum available IOPS (or capacity) of a disk can be determined offline using standard calibration procedures.

Promise: SQLVM promises to reserve for the tenant a certain *IOPS*, denoted $ResIOPS_i$. This promise can again be viewed as a slice of the IOPS capacity available of the underlying physical disk drives. Note that our promise makes no distinction between sequential and random I/Os. The rationale is that even though DBMSs traditionally have developed optimizations for sequential I/O, the stream of I/O requests in a server hosting independent tenant workloads may not be sequential due to the high degree of multiplexing across tenant workloads. However, for tenants whose workloads require scanning large amounts of data (e.g., decision support workloads), the promise can in principle be offered in terms of I/O bandwidth (Mbps). This paper, however, focusses on IOPS.

Metering: The key challenge in metering I/O throughput is in determining if the tenant had "sufficient I/O requests" to meet its reservation and whether the I/O throughput achieved is commensurate with the promise. Similar to CPU utilization, observe that if a tenant had *at least* one I/O request pending, then it had work to utilize the I/O resources. We define the *effective* I/O throughput as the IOPS achieved for the time when the tenant had at least one pending I/O request in the given metering interval. The I/O metering logic flags a violation if the effective I/O throughput is less than $ResIOPS_i$. The rationale and argument for fairness is similar to that in the case of CPU: if requests arrive in bursts, the provider must issue enough I/O requests to meet the effective rate of $ResIOPS_i$, thus preventing the provider from unnecessarily delaying the requests; the provider is not held accountable for periods when the tenant was idle, i.e., did not have any pending I/O requests.

2.3 Memory

While there are many uses of memory in a relational DBMS, we focus here on the two major uses: buffer pool and working memory. The buffer pool is a cache of database pages that is managed using a page replacement strategy (e.g., LRU-k). If a page is not found in the buffer pool, the DBMS incurs I/O to obtain it from secondary storage. Working memory is private to a physical operator used in a query execution plan, such as Hash or Sort. If working memory is limited, the operator may need to spill its state (e.g., partitions of the hash table) to secondary storage, thus again incurring additional I/O. Therefore, promises on memory are also crucial for performance. Similar to static reservation of CPU and I/O capacity, statically allocating a tenant's memory also limits consolidation. Therefore, we seek a way to dynamically distribute memory across tenants, but provide a precise promise to tenants that exposes an illusion of statically-allocated memory.

Promise: To allow dynamic and fine-grained sharing of memory among tenants, our promise is that the number of I/Os incurred in the multi-tenant system is the same *as though* the system had dedicated a certain amount (say 1GB) of buffer pool memory for the tenant; a similar promise applies for working memory. For a given amount of memory M, we define Relative IO as follows:

$$Relative\ IO = \frac{Actual\ IOs - Baseline\ IOs\ (M)}{Baseline\ IOs\ (M)}$$

SQLVM promises a tenant Relative IO ≤ 0 for a given amount of memory. Similar to other resources, a tenant is promised a memory reservation ($ResMem_i$). For example, suppose a tenant is promised a 1GB buffer pool memory reservation. In effect, the promise is that the tenant's workload will see the same *hit ratio* as though a 1GB buffer pool was reserved for the tenant. Similarly for working memory, a promise of 500 MB implies that there would be no more I/O to/from disk for Hash or Sort operators compared to 500 MB of working memory dedicated to that tenant.

Metering: Since memory is allocated dynamically and a tenant's actual memory allocation might differ from $ResMem_i$, the key challenge for metering memory is to determine *Baseline IOs (M)*; *Actual IOs* can be measured directly. This requires a "what-if" analysis to simulate the I/O behavior of the workload as though the tenant had *M* units of memory dedicated to it. The challenge lies in doing this baseline simulation accurately and with low overhead. We have shown (via implementation in Microsoft SQL Azure) that the baseline simulation is feasible in practice and accurate, both for buffer pool memory and working memory. For example, for buffer pool memory, the observation is that the relative I/O is dependent on the page access order, page replacement policy and page metadata (such as dirty bit), and not the actual contents of the pages. The CPU overhead necessary to simulate this baseline buffer pool can be piggybacked on the actual buffer pool accesses and page replacement, and is almost negligible in practice. Finally, we note that if the metering logic determines *RelativeIO* > 0, any such additional I/Os incurred for the tenant are not charged to the tenant's *ResIOPS*; these additional I/Os are charged to the system.

3. IMPLEMENTATION

We have built a prototype of SQLVM inside Microsoft SQL Azure. In particular, we added the ability to specify a SQLVM configuration for a tenant, modified the resource scheduling mechanisms to enable the server to meet reservations of a tenant for each of the key resources: CPU, buffer pool memory, working memory and I/O, and implemented metering logic for each of these resources. In this paper we only discuss the I/O scheduling mechanism and its metering logic. Scheduling mechanisms and metering logic for other resources are beyond the scope of this paper. Note that many database systems already have support for classifying incoming queries and associating them to tenants (e.g., [9]). SQLVM leverages such mechanisms to dynamically determine which tenant issued a query.

3.1 I/O Scheduling

There are three major challenges in implementing I/O scheduling in an RDBMS for meeting *ResIOPS* promised to each tenant. The first challenge is the accuracy and efficiency of the actual scheduling *mechanism*. The second challenge concerns accurately accounting all I/O requests associated with a tenant irrespective of whether an I/O request is directly issued during execution of a tenant's workload or issued by a background system activity on behalf of the tenant. The third challenge pertains to the fact that database systems often use multiple logical drives (*volumes*), e.g., one volume for storing data files and a separate volume for the log file, or data striped across multiple volumes. Therefore, the I/O scheduling mechanism must be able to handle such scenarios.

Scheduling mechanism: A tenant can have multiple queries concurrently executing and issuing I/O requests. These queries can run on multiple cores, and can issue I/O requests independently of other queries belonging to the same tenant. Furthermore, in a multi-core processor, I/O requests of a tenant are not guaranteed to be evenly balanced across cores. Thus, the key challenge in meeting *ResIOPS* accurately across multiple cores and queries is to synchronize a tenant's I/O requests from different cores and from concurrent queries, but with minimal overhead.

Our scheduling mechanism is inspired by the I/O scheduling technique proposed by Gulati et al. [5] for a hypervisor supporting multiple VMs; although there are several new challenges in adapting the technique for a DBMS. In our implementation, we maintain a queue of I/O requests per tenant on each core (as part of the scheduler's data structures). When a tenant's workload issues an I/O request, it is assigned a *deadline* – a timestamp that indicates the time at which the I/O *should be issued* in order for the tenant to meet its *ResIOPS*. Intuitively, if an IO request is issued every T ms, then it results in 1000/T IOPS. For example, if a tenant is promised 100 IOPS, then the system meets the promise by issuing one I/O of the tenant every 10 msec. Thus, deadlines for I/O requests of a particular tenant will be spaced 1/*ResIOPS sec* apart. This deadline assignment requires synchronization across cores. However, this synchronization is lightweight; it requires reading and updating a single tenant-specific counter that is implemented using an atomic operation natively supported on modern hardware architectures. Thus, this mechanism scales well in terms of number of cores and number of concurrent tenants, while providing accurate control over I/O throughput. Once an I/O request is assigned a deadline, it is queued in a pending I/O queue to be issued by the scheduler at the opportune moment.

Whenever a task yields the CPU, the scheduler periodically checks pending I/O requests whose deadline is before *now*. Referring to Figure 2, if *now* = 110, then only Request Id: 1, 3 and 4 are de-queued and issued.

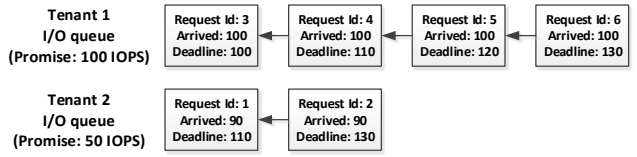

Figure 2. I/O request queue.

Note that in our current implementation, I/O requests for a tenant are issued in the order of arrival. However, since I/O requests can potentially be reordered by the disk controller, preserving the order of I/O requests is not a strict requirement. Therefore, it is possible to reorder I/O requests of a tenant in our queues to achieve higher-level optimizations for tenant workloads. For example, consider a tenant that has issued a short query (requiring only a few I/Os) concurrently with another long-running query (requiring a large number of I/Os) that has already queued a number of I/O requests. In this case, reordering the I/O requests issued by the queries of the tenant can significantly reduce latency of the short query while still achieving the $ResIOPS_i$ for the tenant. Observe also that in addition to meeting $ResIOPS_i$, the above mechanism "shapes" a burst of I/O traffic of a tenant by issuing these requests spaced apart over a period of time. Since a context switch on a scheduler is quite frequent (typically several hundred times a second), fine-grained control over I/O issuance is possible.

Finally, we observe that the DBMS only has control over when a given I/O request is *issued* to the underlying storage subsystem; it does not control when the I/O request *completes*. By controlling the number of concurrent I/O requests issued to the storage subsystem, it is possible to achieve a steady and predictable I/O latency (this

concurrency degree is obtained through typical calibration techniques). In practice, we observe that in the steady state, the rate of requests issued is same as that completed. However, in the strictest sense, this algorithm only promises a reservation on I/O requests per second *issued* by the database server on behalf of the tenant.

Accurately accounting direct and indirect I/Os: I/Os issued by a tenant's workload can conceptually be categorized as *direct* – i.e., issued during the execution of the workload, or *indirect* – i.e., issued by a system thread on behalf of the tenant as part of a background activity. Examples of direct I/Os are reads of data pages required for query execution, log writes, and reads and writes performed during a backup or restore database operation. Examples of indirect I/Os include: flushing dirty data pages to the disk, checkpointing, and data replication for availability. Direct I/Os are readily accountable and can be directly associated with the tenant that issued the I/O. However, since indirect I/Os are issued from a system thread, additional information must be extracted from the context to identify which tenant the I/O should be accounted to. For example, for the system thread that lazily flushes dirty buffer pool pages, we look up the file being written to, and from the file identify the tenant database to whom the page belongs. Capturing all indirect I/Os requires similar modifications to multiple components within the database engine.

***Governing multiple logical drives (volumes)*:** A logical drive, or *volume*, is a collection of disk spindles (or an SSD) that is exposed to the OS as single device. A file on that drive is typically striped across all spindles of that drive. Moreover, DBMSs often use multiple volumes; a typical configuration might be to use one volume for the data file(s) of a database and one volume for the log file. In SQLVM, we govern each such volume independently. Thus, an IOPS reservation for a tenant internally maps to an IOPS reservation *per volume*. For the scheduling mechanism described above, this implies that we need to maintain one logical queue of I/O requests per tenant per volume.

3.2 I/O Metering

We describe the I/O metering logic using a running example. Consider a reservation of 100 IOPS for a tenant. This promise can have two different interpretations, and therefore two different metering schemes. The *strong* version is that as long as the tenant has at least one I/O request pending, the system will issue one I/O every 10 *msec* (i.e., 1/100 sec). Such a promise might be attractive to applications striving for low latency in addition to low variability in I/O throughput. For this interpretation, the metering logic computes the delay (*d*) between when the I/O should have been issued and when it is actually issued (see Figure 3). The scheduling mechanism described in Section 3.1 maintains a *deadline* for each I/O that identifies the time by when the I/O should be issued to achieve the desired *ResIOPS*. Thus, when an I/O request is actually issued, the metering logic uses the deadline to compute the delay (if any). At the end of the metering interval, a distribution of these delays is reported (e.g., maximum, average, percentiles) that quantifies the violation of the promise. Note that if delay (*d*) is 0 for every I/O request, then *ResIOPS* promise is met.

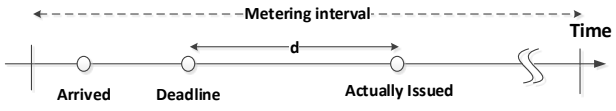

Figure 3. I/O Metering for reserved IOPS.

A *weaker* version promises that the *average* I/O throughput over the meeting interval is at least 100 IOPS. This interpretation relaxes the requirement to issue one I/O every (*1*/*ResIOPS*) sec as long as the average rate for the interval meets the promised *ResIOPS*. Metering for the weaker version also leverages the *deadline*. At the end of the interval, each I/O request whose deadline lies within the interval but was *not* issued represents a violation. If there are *n* such violations in a metering interval of *t* sec, then the promise is violated by *n*/*t* IOPS. Deadline assignment inherently factors out idle time and hence this metering logic is equivalent to that described in Section 2.2.

4. EXPERIMENTS

We present an evaluation of the SQLVM prototype implemented in Microsoft SQL Azure. The goal of this evaluation is to demonstrate the following: (i) when resources are not overbooked, SQLVM is able to allocate resources as promised to a tenant, even when many other tenants with resource-intensive workloads are concurrently executing on the database server and contending for the resources; (ii) when promises on key resources are met, using SQLVM results in considerably better performance isolation (in terms of throughput and response times of the tenant's workload) compared to other alternative approaches; and (iii) when resources are overbooked and reservations cannot be met, our independent metering logic detects these violations in the promised reservations.. Our experiments demonstrate how metering can establish auditability and accountability of the service provider to the tenant.

4.1 Experimental Setup

Our evaluation uses a workload suite consisting of four different workloads that represent diversity in resource requirements: TPC-C [12] and Dell DVD Store [4] benchmarks are OLTP-style workloads; TPC-H [13] benchmark is a DSS-style workload; and a synthetic micro-benchmark (called CPUIO) that generates queries that are CPU- and I/O-intensive.

The TPC-C benchmark consists of nine tables and five transactions that portray a wholesale supplier. The five transaction types in TPC-C represent a supplier's business needs and workloads. A typical TPC-C workloads represents a good mix of read/write transactions where more than 90% of transactions have at least one write operation (insert, update, or delete).

The Dell DVD Store benchmark represents an e-commerce workload where transactions represent typical user-interactions such as logging-in, browsing some products, adding items to an order, and purchasing the order. This benchmark generates a good mix of read and write transactions.

The TPC-H benchmark is a decision support benchmark that consists of a set of twenty two business-oriented ad-hoc queries. This workload simulates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

Finally, the CPUIO benchmark comprises of a single table with a clustered index on the primary key and a non-clustered index on the secondary key. The workload consists of three query types: (i) a CPU-intensive computation; (ii) a query involving a sequential scan with a range predicate on the primary key of the table; and (iii) a query with a predicate on the non-clustered index which performs random accesses to the database pages.

Each tenant connects to a separate database and executes an instance of one of these workloads. The tenants are hosted within a single instance of the database server with a 12-core processor (24

logical cores with hyper-threading), data files striped across three HDDs, the transaction log stored in an SSD, and 72 GB memory.

4.2 Meeting Reservations

In this controlled experiment, we use a micro-benchmark to evaluate SQLVM's ability to meet the resource reservations when enough resources are available at the database server, i.e., the resources are not overbooked. We focus on the I/O throughput and CPU utilization. We focus on one resource-at-a-time to rule out any interactions between resources. We co-locate two tenants executing *identical* workloads but with different resource reservations set to their corresponding SQLVMs. We use the CPUIO workload to generate CPU- and I/O-intensive workloads. This experiment also demonstrates SQLVM's ability to dynamically adjust the resource reservations of a tenant.

In the first part of the experiment, we focus on the I/O throughput. The CPUIO benchmark only issues the I/O-intensive queries with minimal CPU requirements. Tenant T_1's SQLVM is configured with $ResIOPS_1$ = 60 and T_2's configuration is $ResIOPS_2$ = 20. Figure 4 plots the actual I/O throughput achieved by each tenant. The figure shows that both tenants were allocated their promised I/O throughput. For instance, in the first 300 seconds of the experiment, T_1 received 63 IOPS on average with a standard deviation of 6.9, and T_2 received 20 IOPS on average with a standard deviation of 3.8. After 300 seconds into the experiment, we switch the IOPS reservation for the tenants, i.e., $ResIOPS_1$ = 20, $ResIOPS_2$=60). Figure 4 shows that the IOPS achieved also switches almost instantaneously.

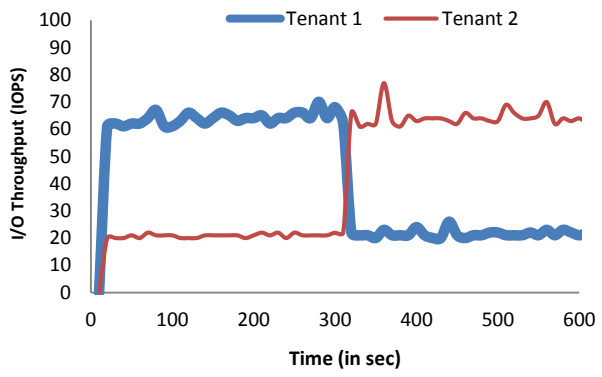

Figure 4. Tenants' I/O throughput.

Although this experiment focused on two tenants, our evaluation has shown that SQLVM continues to meet the promises even as more tenants are added and the system has enough resources to meet the promises. In an experiment with 25 tenants executing CPU-bound workloads, where $ResCPU_1$ = 50%, T_1's actual average CPU utilization was 49.5%. Tenants other than T_1 did not have any CPU reservations.

4.3 Performance Isolation

In this experiment, four tenants concurrently execute one instance of each workload in our suite of benchmarks. We ran different workload combinations; for brevity, this experiment only reports experiments with T_1 executing the TPC-C workload and other tenants are executing the remaining three workloads. We compare three different configurations for the tenant (T_1) of our interest: (i) T_1 is promised a resource reservation, i.e., executing within a SQLVM; (ii) T_1 is promised only a maximum limit on resources with no reservations (represented as *Max-only*); and (iii) no promises on resources, i.e., a "best-effort" *Baseline* server.

In the SQLVM configuration, T_1 is promised $ResCPU_1$ = 50%, $ResIOPS_1$ = 200, and $ResMem_1$ = 2GB. The remaining tenants do not have any reservations; however, they have a maximum limit of 50% CPU utilization, 250 IOPS, and 2GB memory. In the Max-only configuration, all tenants have a maximum limit of 50% CPU utilization, 250 IOPS, and 2GB memory. The Baseline has no reservations or maximum limits. The experiment starts with T_1 as the only active tenant; a new tenant workload is added every five minutes. When all four tenants are active, the system is overbooked with respect to maximums but had enough resources to meet the promised reservations.

Tenant T_1's workload is CPU-bound; T_1 is executing the TPC-C workload where the database (10 warehouses, about 1GB) fits into the buffer pool allocated. Therefore, the end-to-end performance, in terms of throughput and query response times, depends on the CPU allocated to each tenant. SQLVM allocated 49% CPU to T_1 on average, while the average CPU allocation for Max-only and Baseline are 23% and 27% respectively. Since the system was overbooked based on the max, the Max-only promise results in lower actual utilization. Moreover, Baseline results in higher utilization since T_1 can potentially use more than 50% CPU at certain time instances while Max-only always limits utilization to 50%; however, the actual utilization achieved in the two latter combinations depends on the other tenants' workloads.

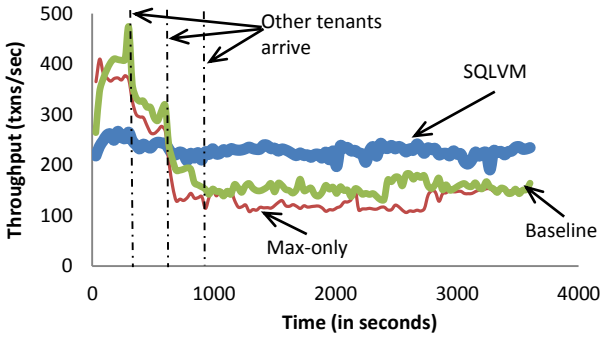

Figure 5. T_1's throughput as three other tenants are added.

Resource isolation in SQLVM in turn results in significantly better performance isolation compared to the alternative configurations. Figure 5 plots the throughput of tenant T_1 (aggregated over a 30 second period) during the experiment; time since the start of the experiment is plotted on the *x*-axis. The three vertical lines denote the time when other tenants start executing their workloads. T_1, executing within a SQLVM, observed minimal impact on throughput as other tenants are added; SQLVM ensures that T_1 continues to receive the resources promised. Max-only and Baseline, on the other hand, result is a significant impact on throughput (about 40% lower compared to SQLVM) since the actual resources allocated to T_1 differ depending on the resource demands of the other tenants. Further, in an experiment where 24 other tenants were added (one every minute) to the system, T_1's throughput was *6X* higher when executing within a SQLVM than compared to the Max-only or Baseline configurations. In general, by adding more tenants, it is possible to make the performance degradation in Max-only and Baseline arbitrarily bad.

Table 1 reports the 99th percentile response times and its standard deviation for T_1 for the experiment with 24 other tenants. This further demonstrates SQLVM's superior isolation of T_1 from the

other tenants' workload. Specifically, for all TPC-C transaction types, T_1's 99th percentile response time is about *4X* lower than that of the alternatives, while also having a lower standard deviation in response times. Such low variance in response times is remarkable considering that the server's overall average CPU utilization was more than 95% throughout the experiment.

Table 1. The 99th percentile and standard deviation of T_1's end-to-end response time (in ms) with 24 other active tenants.

Operation	SQLVM		Max-Only		Baseline	
	99th %	Std. Dev.	99th %	Std. Dev.	99th %	Std. Dev.
Delivery	935	181	4416	953	4056	1532
NewOrder	619	202	2762	672	2589	761
OrderStatus	113	3327	421	6392	390	4989
Payment	327	246	1042	358	1170	235
StockLevel	2437	2812	22580	5675	5897	2056

4.4 Validating Metering

Experiments reported in the earlier sections focused on scenarios where the server had sufficient resources to meet the promised reservations. That is, the sum of all the reservations did not exceed the available resources at the server. However, cloud infrastructures often overbook resources to minimize their operating costs. A unique aspect of SQLVM is the independent metering mechanisms to establish accountability in such situations. Metering provides the ability to detect any violations in a tenant's promise. In this experiment, multiple tenants are promised reservations such that the server is overbooked. The goal is to observe whether metering accurately determines violations.

In this experiment, eight tenants are co-located at the same SQL Server instance. Each tenant is promised $ResIOPS_i = 80$. Therefore, the aggregate of all I/O reservations is 640 IOPS which is more than double of the approximately 300 IOPS capacity of the underlying disk sub-system. That is, this server is overbooked on I/O throughput. Each tenant is an instance of one of the four workloads in our benchmark suite; each workload is executed by exactly two tenants. The tenant workloads are configured to be I/O-intensive; the server has enough CPU to process the queries in the workloads and generate the I/O requests.

Figure 6 plots the number of read I/O requests for two of the eight tenants: T_1 is executing the TPC-C workload and T_2 is executing the Dell DVD store workload (DS2). As is evident from the figure, after about 300 seconds into the experiment, when the fourth tenant starts executing its workload, the read I/O throughput achieved by both tenants is lower than their respective reservations, which is 80 IOPS. Therefore, the tenants' promises might potentially be violated, provided the tenants had sufficient pending I/O requests to meet the reservation. Figure 7 plots the I/O violations for the two tenants reported by the I/O metering logic. The *y*-axis of Figure 7 plots the number of I/O operations that were tagged to be issued in a given metering interval but were not issued due to insufficient capacity of the disk sub-system. As is evident from the figure, I/O violations are reported for both tenants.

The metering logic's ability to differentiate between a violation and insufficient requests by the tenants' workload is also evident from Figure 7. Even though both tenants were promised 80 IOPS and T_2 achieved a lower read I/O throughput compared to T_1, the IOPS violated for T_1 is typically higher than that of T_2. This implies that T_1 is more I/O-intensive that T_2 and T_1's workload generated a higher demand for I/O. That is, one of the contributing factors for T_2's lower I/O throughput is the insufficient number of I/O requests. If we just considered the IOPS achieved, as shown in Figure 6, one may incorrectly conclude T_2's promised reservation was violated to a greater extent. This ability to differentiate these two scenarios is critical for a relational DaaS provider that serves a variety of workloads and does not control which queries the tenants will execute. Therefore, metering provides a mechanism to audit the scheduling algorithms, if needed, and establish accountability.

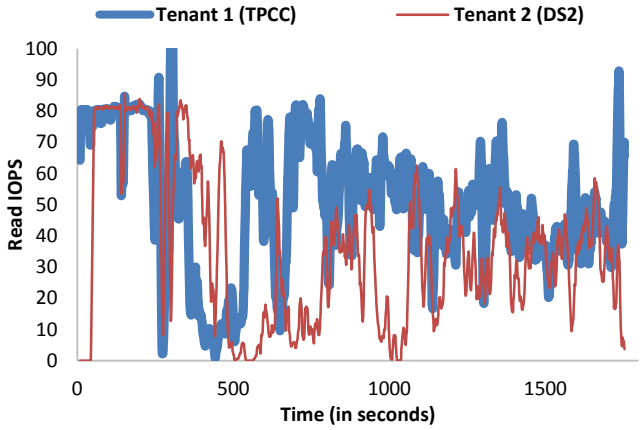

Figure 6. Tenant's read I/O throughput.

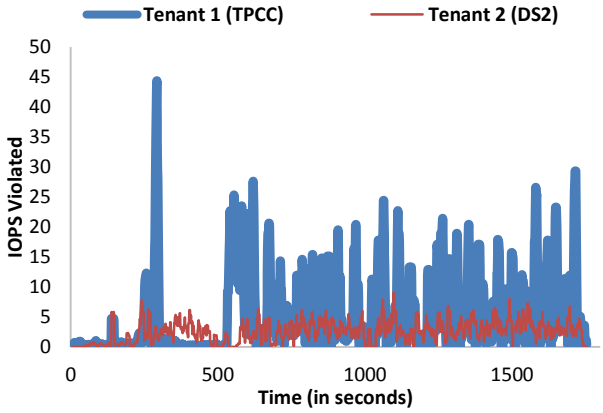

Figure 7. IOPS violated due to overbooking of the I/O capacity.

Note that in this experiment, we considerably overbooked the server to magnify the effect of overbooking and validate the metering logic. In practice, a service provider can be intelligent in determining which tenants to co-locate such that even when a server is overbooked, the chances of a violation is low. A provider can leverage the fact that many tenants often do not have enough work to utilize all resources reserved. Such techniques are orthogonal to SQLVM mechanisms and are interesting directions of future work.

5. RELATED WORK

From a tenants' perspective, performance isolation in the form of workload-level service-level agreements (*SLAs*), such as queries per second or end-to-end query latency, would be ideal. For instance, Chi et al. [2] proposed using piecewise-linear latency SLAs for differentiated service and scheduling of queries in a batch-oriented system. However, as pointed out earlier in this paper, a relational DaaS platform such as Microsoft SQL Azure, must support flexible, parameterized, and often *ad-hoc* SQL queries. In such a setting, *robust* estimation of the expected

response time, resource requirements, and progress of an arbitrary SQL query remains an open and challenging problem. Therefore, it is extremely hard for a provider to guarantee a latency (or throughput) SLA with a high confidence. Moreover, tenants' workload, data access distributions, and query mixes may change over time. As a result, supporting latency SLAs even for parameterized SQL queries is also challenging. Finally, even when a high-level SLA is exposed, tenants will eventually share resources at a server. Therefore, a fine-grained resource sharing abstraction, such as SQLVM, is important to allocate appropriate resources to a tenant's workload in the presence of other contending workloads.

Curino et al. [3] and Lang et al. [7] approach consolidation of multiple databases in a single server by analyzing the workloads, identifying how these workloads interact with one another, and recommending which databases should be co-located in order to meet performance goals (or SLO – Service-Level-Objectives). Xiong et al. [15] constructs machine learning models to predict query performance as a function of resources allocated to it, and then use such models to allocate resources so that query latency SLO can be met. SQLVM is complementary to these approaches since it provides resource-level isolation from other tenants, and makes no assumptions about the specific workloads of tenants. SQLVM can potentially be used as a building block to build such recommenders, since SQLVM can ensure that the tenants are actually allocated the resources that the models assume.

Armbrust et al. [1] propose a SQL-style data independence layer on top of a key-value store for achieving the SLO of predictable response times for queries. Their model limits the kinds of queries that users can pose, which in turn enables bounding the work done per query. In contrast, SQLVM provides assurances at the level of resources, but does not impose any restrictions on the queries that tenants can execute.

Resource management has been supported in many commercial relational DBMSs for a while. However, such support has been limited to providing a maximum limit on resource utilization, assigning affinities between resources and tenants (such as affinitizing one or more cores to tenants), or throttling runaway queries. As shown in our experiments, such Max-only approaches are not enough for providing performance isolation while supporting high consolidation factors. Reserving resources with a minimum promise for a tenant and then metering the promise to establish accountability are SQLVM's novel contributions.

The resource reservation abstraction has also been proposed in the context of operating systems and shared computational grids. For instance, Mercer et al. [8] proposed supporting processor capacity reserves in operating systems for real-time multimedia applications. Similarly, Smith et al. [10] proposed the concept of reservations in shared grid computing systems. SQLVM differs from these approaches by presenting an abstraction for resource reservations at a fine granularity within a DBMS (unlike coarse-grained reservations in large shared grids used predominantly for batch-oriented jobs) and without requiring any advance knowledge about the workload or requiring workloads to have certain behavior (unlike real-time multimedia systems).

Note that for a variety of reasons (e.g., see [11]), DBMSs typically need to assume control of most resources, and therefore cannot benefit from such OS- or hypervisor-level mechanisms. Furthermore, database workloads and the DaaS context bring unique challenges that require us to rethink the assurances and the mechanisms necessary to support them within the DBMS. Therefore, it is critical to provide an abstraction for fine-grained resource sharing and isolation such as SQLVM.

Finally, there has been extensive work in area of *workload management*, particularly in a traditional data warehouse setting where queries can be resource intensive (Krompass et al. [6] present an overview). We believe that SQLVM can be valuable even in such traditional enterprise scenarios since it can be used to more tightly control resource allocation to different classes of queries.

6. CONCLUDING REMARKS

We presented SQLVM, a lightweight abstraction of a VM running within a database server that provides resource reservations. Implementing this abstraction requires new fine-grained resource allocation mechanisms aware of reservations, in conjunction with metering logic that audits whether or not the promise is met. Our implementation in Microsoft SQL Azure demonstrates the value that SQLVM provides in resource and performance isolation.

One important area of future work in SQLVM arises for the case of overbooking, where the mechanisms and policies used by the system need to be able to trade-off between multiple criteria such as reducing penalties due to violations of the promise, and fairness to tenants. Finally, while SQLVM already manages several key DBMS resources, there is an opportunity for even greater performance isolation by extending SQLVM to other resources such as network bandwidth and data/instruction cache. The feasibility of such extensions requires further investigation.

7. ACKNOWLEDGMENTS

Feng Li, Hyunjung Park, Vamsidhar Thummula, Willis Lang, and Hamid Mousavi have contributed significantly to the SQLVM project when they visited Microsoft Research. Several members of the Microsoft SQL Azure product group, including Peter Carlin, George Reynya, and Morgan Oslake, have provided valuable feedback that has influenced our work. The authors would also like to thank Christian Konig, Ishai Menache, Mohit Singh, Johannes Gehrke, and the anonymous reviewers for their insightful comments on earlier drafts of this paper.

8. REFERENCES

[1] Armbrust, M., Curtis, K., Kraska, T., Fox, A., Franklin, M. J., and Patterson, D.: A. PIQL: Success-Tolerant Query Processing in the Cloud. In *Proc. of the VLDB Endowment, 5(3), pp. 181-192*, 2011.

[2] Chi, Y., Moon, H. J., and Hacigumus, H.: iCBS: Incremental Cost-based Scheduling under Piecewise Linear SLAs. In *Proc. of the VLDB Endowment, 4(9), pp. 563-574*, 2011

[3] Curino, C., Jones, E. P. C., Madden, S., and Balakrishnan, H.: Workload-aware database monitoring and consolidation. In Proc. of the ACM SIGMOD Int. Conf. on Management of Data, pp. 313-324, 2011.

[4] Dell DVD Store Database Test Suite: The DVD store version 2 (DS2), v2.1, http://linux.dell.com/dvdstore, Retrieved: Nov 20th, 2012.

[5] Gulati, A., Merchant, A., and Varman, P. J.: mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation (OSDI), pp. 437-450, 2010.

[6] Krompass, S., Scholz, A., Albutiu, M, Kuno, H. A., Wiener, J. L., Dayal, U., and Kemper, A.: Quality of Service-enabled

Management of Database Workloads. In *IEEE Data Eng. Bull. 31(1), pp. 20-27*, 2008.

[7] Lang, W., Shankar, S., Patel, and J., Kalhan, A.: Towards Multi-Tenant Performance SLOs. In Proc. of the 28th IEEE Int. Conf. on Data Engineering (ICDE), pp. 702-713, 2012.

[8] Mercer, C., Savage, S., and Tokuda, H.: Processor Capacity Reserves: Operating System support for Multimedia Applications. In Proc. of the Int. Conf. on Multimedia Computing and Systems, pp. 90-99, 1994.

[9] Resource Governor, Microsoft SQL Server 2012. http://msdn.microsoft.com/en-us/library/bb933866.aspx, Retrieved: Nov 20th, 2012.

[10] Smith, W., Foster, I. T., and Taylor, V. E.: Scheduling with Advanced Reservations. In Proc. of the 14th Int. Parallel and Distributed Processing Symposium (IPDPS), pp. 127-132, 2000.

[11] Stonebraker, M. Operating System Support for Database Management. *Communicatons of the ACM*, 1981.

[12] Transaction Processing Performance Council, TPC-C Benchmark, v5.10, http://www.tpc.org/tpcc, Retrieved: Nov 20th, 2012.

[13] Transaction Processing Performance Council, TPC-H Benchmark, v2.10, http://www.tpc.org/tpch, Retrieved: Nov 20th, 2012.

[14] Urgaonkar, B., Shenoy, P. J., and Roscoe, T.: Resource Overbooking and Application Profiling in Shared Hosting Platforms. In Proc. of the 5th USENIX Symp. on Operating System Design and Implementation (OSDI), 2002.

[15] Xiong, P., Chi, Y., Zhu, S., Moon, H. J., Pu, C., and Hacigumus, H.: Intelligent management of virtualized resources for database systems in cloud environment. In Proc. of the 27th IEEE Int. Conf. on Data Engineering (ICDE), pp. 87-98 2011.