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

Academic Critique

Chris Dusyk
200268705

Abstract

Relational Database-as-a-Service (DBaaS) can share database server resources among multiple tenants, called multi-tenancy. This enables cost reduction for the cloud service provider, however this can impact performance due to resource demands on other tenants. “SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service”, by Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri, explores multi-tenancy in the database service and proposes low-overhead techniques to meter resource allocation. This paper will explore multi-tenancy, the proposed SQLVM, and critique the paper provided by Narasayya et al.

1. Introduction

To start with, this paper will explain some of the terms that will be used throughout. A **tenant** is work thread for a client, or group of clients, making service requests of a hosted service. **Multi-tenant** refers to a system that has many tenants making resource requests of a hosted service. Most DBaaS providers are multi-tenant systems, such as Microsoft SQL Azure and AWS RDS.

Multi-Tenancy gives DBaaS providers the ability to balance workloads for one customer across multiple tenants. For example, one tenant (customer) with a heavy workload will have their resource needs shared by a few tenants. This reduces the performance hit for the one tenant, however has the potential to impact the performance of the tenants the workload is distributed to, which can negatively impact other customers. These shared resources make it easy for providers to keep hosting costs down, since it means they require less of everything, but it means that there is no absolute guarantee that a tenant will receive all of the resources it needs. To resolve this, providers will enforce maximum resource usage on shared resources for each tenant, however this can lead to providers to overbook resources (promise more resources in aggregate to tenants than the available system capacity) (Urgaonkar et al, 2002). Another common method of managing shared resources among tenants is a reservation system, where tenants can reserve certain amounts of resources for a specific block of operations. For example, Tenant1 can reserve 100 I/Os per second (IOPS), under the promise that Tenant1 will assume a workload value of 100 IOPS regardless of what other tenants are concurrently executing.

The most critical component of resource allocation is metering promised resources. The system must be auditable so that if the resource promise is not met, the provider can determine if this is caused by a tenant not using its promised resources or attempting to use more than its promised resources. Typically, a multi-tenant system will have hundreds of tenants, so metering resources is critical to ensure tenants receive sufficient resources while keeping overhead costs low (Narasayya et al, 2013).

Another factor to consider in a multi-tenant system is that the database server itself consumes some resources for necessary system tasks. The proposed SQLVM treats these system tasks as an “internal tenant” with certain resource requirements (Narasayya et al, 2013).

A concern with any multi-tenant system is the potential to overbook all resources. This is generally a rare occurrence, but the potential to happen exists. In this scenario, additional policies are required to determine which tenants’ promises will be violated. Factors in these policies include: fairness to tenants, and the need to minimize penalties incurred when the promise is violated.

2. Multi-Tenancy

There are two main architectures for hosted services: multi-tenant, and multi-instance. multi-instance splits tenant requests across many small, isolated service instances. In contrast, multi-tenant systems have many tenants served by a larger service instance. In this case, this paper refers to DBaaS services. A multi-tenant DBaaS service uses a more powerful database server that serves many, typically hundreds, of tenants.

Multi-tenant systems follow a basic architecture seen in Figure 1, just below. The tenants sit in the datacentre network, which pushes requests to the database server process. The database server process controls which tenant needs to connect to which tenant database, and passes the requests to the correct database (Lang, 2012). Once passed to the tenant database, the database engine manages the data retrieval and modification. Server resources are shared across all tenants maintained on the server, which will be discussed in detail in section 3.

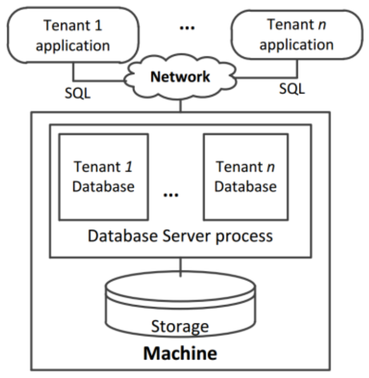


Figure 1. A multi-tenant database system.

Image source: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

2.1 Benefits

There are many benefits to multi-tenant systems, including: cost savings, cheaper data aggregation, and simplified release management. To start with, multi-tenant systems provide cost savings by reducing the memory and processing overhead that having many virtual systems creates. This reduces the number of software licences and required system resources needed overall.

2.2 Disadvantages

The development effort in a multi-tenant system is much higher, due to the additional complexity involved in managing per-tenant metadata. This can be likened to developing multi-threaded vs single-threaded systems. While the complexity is not a major barrier, it does drive up the development effort and cost to the provider. This must be balanced against the cost savings from the reduced resource overhead.

The biggest potential disadvantage, however, is resource constraints. A multi-tenant system relies on splitting the available pool of resources across all tenants. The fair splitting and management of these resources is a major challenge facing multi-tenant design and architecture currently. The paper discussed here goes into detail on the proposed SQLVM (SQL Virtual Machine) proposed by Microsoft's SQL Azure team, to manage resources.

2.3 Multi-Tenant Architecture Approaches

There are three main approaches to managing multi-tenant data: separate databases, shared databases – separate schemas, and shared databases – shared schemas.

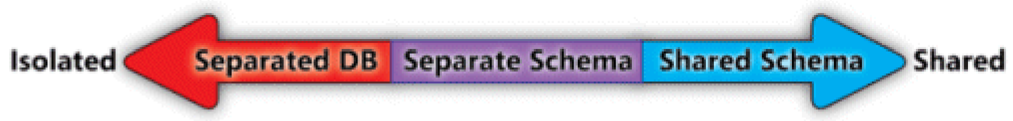


Figure 2. Image source: https://msdn.microsoft.com/en-us/library/aa479086.aspx

2.3.1 Separate Databases

Each tenant has their own database in this approach. This is, by far, the simplest approach. While processing resources are shared amongst all tenants on the server, each tenant has its own isolated database. Metadata associates each database with the correct tenant, and database security prevents any tenant from accidentally or maliciously accessing another tenant's data (Chong, 2006).

Giving each tenant their own database makes it easy for the tenant to manage their data with no impact to the data of any other tenant. It also makes it much easier to manage backups. Each tenant's database is backed up normally and there's no extra development effort required by the provider to separate data for each tenant for the backups. Likewise, restoring from backups is quicker and easier.

The main disadvantage to this approach is the increase in hardware and maintenance requirements and costs. This requires more database servers, which requires more servers, disks, and licences. It is, however, ideal for customers who require added security and customizability. An example of this would be banking or medical systems, which typically have very high data isolation requirements.

2.3.2 Shared Database, Separate Schemas

The most common approach is to use shared databases, with separate schemas. Each tenant's tables are grouped into the tenant's schema, with many tenant schemas residing in the same database. This approach is also easy to develop and maintain, but does reduce the hardware and maintenance costs in comparison to the separate databases approach. Having several tenants per database reduces the amount of hardware and licencing required, making it cheaper for both the provider and the tenant. Each tenant's data is still isolated, though not as strongly. This would not be ideal for high-risk data, but would be appropriate for a small business. Typically, this is ideal for a tenant with hundreds of tables at most.

The biggest drawback to this approach is that it is significantly more difficult to restore a tenant's data in the event of a failure or corruption. In most Database Management Systems (DMBS), restoring a specific schema is not a simple task, especially in comparison to restoring an entire database.

2.3.3 Shared Database, Shared Schemas

This approach uses the same database and the same set of tables to host multiple tenants' data. A given table can include records from multiple tenants with a TenantID column to associate records with the tenant that owns the data. This approach has by far the lowest cost to the provider, and the lowest hardware costs. However, having multiple tenants share the same tables, this approach can easily incur extra development effort to ensure security and data isolation. In particular, there will be huge development effort to ensure that a tenant bug or mistake won't damage other tenants' data. The data isolation in this approach is minimal, with fairly high risk. Backing up and restoring data becomes much more complicated as well. If one tenant's data needs to be restored, the process involves selectively deleting and inserting rows with the correct TenantID, which can be risky to other tenants' data. While these are serious concerns, the most obvious issue to the tenants themselves is the potential performance issue. With many tenants issuing requests to the same tables, performance cannot be guaranteed. One tenant selecting or writing a large amount of data can cripple other tenants' performance.

This approach only really works for clients looking for very cheap data hosting, who are not concerned with data isolation and performance.

3. SQLVM

The Microsoft SQL Azure team proposed the SQLVM as a method of reserving key resources in a database system. The idea behind SQLVM is to provide a lightweight Virtual Machine (VM) template that provides resource isolation across tenants. A key part of the SQLVM is the metering logic that audits tenants' promised resources and their use. 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 (Narasayya et al, 2013). This is particularly important to both clients and providers to establish accountability and service guarantees. One of the key challenges in metering is the nature of database access. Database requests often come in bursts, making it difficult to predict and allocate resources efficiently and provide fair allocation to all tenants.

There are three main resources that need to be allocated amongst tenants: CPU processing, I/O, and memory. These resources need to be shared by the tenants, which is where metering becomes important.

3.1 CPU

Database servers run on processors with many cores, using hyper-threading. On each core, a scheduler decides which queued task gets to run on that core next. For a tenant and a given core, the *CPU Utilization* over an interval of time is the percentage of time of which a task of that tenant is running on the core. So, for k cores, the total time the tenant's tasks run across all cores is a percentage of $(k * time\ interval)$.

3.1.1 CPU Promise

For tenant T_i, SQLVM will reserve a certain CPU Utilization, denoted by $ResCPU_i$. This means that T_i is promised a slice of the CPU time on available core(s), but does not mean that the tenant will get a statically allocated core to itself. This allows better use of resources as SQLVM 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 (Narasayya et al, 2013).

3.1.2 CPU Metering

The main challenge in CPU metering is in defining the concept of sufficient work for a tenant. If a tenant has at least one task that is running or is runnable, then it has work that can use the CPU. As such, the metering problem is 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, otherwise the provider has violated the promise (Narasayya et al, 2013). For example, if T_1 was promised $ResCPU_1 = 10\%$ and if T_1 had at least one task ready to run for 500ms, the provider violates the promise only if the allocated CPU is less than 50ms. This is fair because the provider is not held accountable for the tenant being idle.

3.2 I/O

Database workloads require adequate I/O throughput (IOPS) and I/O bandwidth (bytes/sec). As seen with CPU allocation, dedicating static sets of disks per tenant to achieve acceptable IOPS heavily limits the amount of consolidation and efficiency, as well as raises hardware costs. Therefore, sharing IOPS available to each disk is important. The following sections focus primarily on I/O throughput, but also apply to I/O bandwidth.

3.2.1 IOPS Promise

For tenant T_i SQLVM will reserve a certain IOPS, denoted by $ResIOPS_i$. This promise represents a slice of the IOPS capacity available of the underlying disks. This does not make a distinction between sequential and random I/O's, on the grounds that DBMSs have developed their own optimizations for handling sequential and random I/O's, so there is no point in adding another layer to it.

3.2.2 IOPS Metering

The main challenge in metering I/O throughput is determining if the tenant has sufficient I/O requests to meet its reservation, and whether the I/O throughput achieved matches the promise. Much like CPU Utilization, if a tenant has at least one I/O request pending then it has work to utilize the I/O resources.

Effective I/O throughput is the IOPS achieved for the time when the tenant had at least one pending I/O request in the metering interval. The metering logic flags a violate if the effective I/O throughput is less than $ResIOPS_i$. If requests arrive in bursts, the provider must issue enough I/O requests to meet the effective rate of $ResIOPS_i$ to prevent unnecessarily delaying requests. As with the CPU metering, the provider should not be held accountable for when the tenant is idle.

3.3 Memory

This paper focuses on the two major uses of memory for a DBMS, namely the buffer pool and working memory. The buffer pool acts as a cache of database pages, managed by using a page replacement strategy (such as LRU-k) (Narasayya et al, 2015). If a page is not found in the buffer pool, the DBMS must perform an I/O operation to obtain it from the physical disk. Working memory is used by a physical operator in a query execution plan, such as a Hash or Sort. If working memory is limited, the operator may need to spill its state to secondary storage, thus again incurring additional I/O (Narasayya et al, 2013).

Similar to statically allocating CPU and I/O capacity, statically allocating a tenant's memory limits consolidation and increases resource requirements and costs. Dynamically allocating memory across tenants is ideal, but there is a need to promise tenants a certain amount of memory in a way that appears statically-allocated.

3.3.1 Memory Promise

The SQLVM proposes to allow dynamic sharing of memory among tenants by promising that the number of I/Os incurred in the multi-tenant system will be the same as though the system had dedicated a certain amount of buffer pool memory. The same promise applies to working memory as well. For a given amount of memory M relative IO is defined as

$$RelativeIO = \frac{ActualIOs - BaselineIOs(M)}{BaselineIOs(M)}$$

SQLVM promises a tenant $RelativeIO \leq 0$ for a given amount of memory. A tenant T_i is promised a memory reservation of $ResMem_i$. For example, a tenant is promised a 1GB buffer pool 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 (Narasayya et al, 2013).

3.3.2 Memory Metering

Given that memory is allocated dynamically, and a tenant's actual memory allocation might differ from $ResMem_i$, the main challenge is to determine the $BaselineIOs(M)$. $ActualIOs$ can be easily measured, however there will be some analysis required to calculate a $BaselineIOs(M)$. The challenge is in doing the simulation accurately. Narasayya et al, 2013, showed that the baseline simulation was feasible and accurate in Microsoft SQL Azure, both for buffer pool memory and working memory.

4. Implementation

For the paper, Narasayya and his team built a prototype of the SQLVM inside Microsoft SQL Azure. As Microsoft employees, they were able to specify the configuration for the database tenants and modify the resource scheduling mechanisms to enable the server to meet the reservations of each tenant. They were also able to implement the metering logic for each resource, thus creating the SQLVM implementation they proposed.

4.1 I/O Scheduling

There are three difficulties in implementing I/O scheduling to meet the $ResIOPS$ promised to each tenant. The first challenge is the accuracy and efficiency of the mechanism. The second challenge is to accurately account for all I/O requests associated with a tenant, irrespective of where the I/O request was issued from. The third challenge is related to DBMS's using multiple logical drives, with data striped across many of the drives.

4.1.1 Scheduling Mechanism

A tenant can have multiple queries executing and issuing I/O requests concurrently. This impacts both IOPS and CPU usage. On a multi-core processor, I/O requests are not guaranteed to be evenly balanced across cores. The challenge in meeting $ResIOPS$ is to synchronize a tenant's I/O requests from different cores and from concurrent queries with minimal overhead.

In this implementation, they maintain a queue of I/O requests per tenant on each core. 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 $ResIOPS$). For example, if an I/O request is issued every Tms, then it results in $\frac{1000}{T}$ $IOPS$. The deadlines for I/O requests of a particular tenant will be spaced $\frac{1}{ResIOPS}$ sec apart. This requires synchronization across all cores, otherwise little there will be little advantage to this approach. Thankfully, the synchronization is very lightweight as it makes use of native atomic operations on modern hardware architectures.

Whenever a task yields the CPU, the scheduler checks pending I/O requests whose deadline is before the current time. Narasayya notes that in this implementation, I/O requests for a tenant are issued in the order of arrival. However, since I/O requests can be potentially reordered by the disk controller, preserving the I/O requests is not a strict requirement. This means it is possible to reorder the I/O requests of a tenant to reach a better optimization for the tenant workloads. In addition to helping meet $ResIOPS_i$, this helps shape a burst of I/O traffic of a tenant by issuing the requests spaced apart over a period of time.

4.1.2 Accurately Counting Direct and Indirect I/O

I/Os issued by a tenant's workload fall into two categories: direct (issued during the execution of the workload), and indirect (issued by a system thread on behalf of the tenant as part of background activity). Direct I/O requests are easy to account to and are directly associated with the tenant that issued the request. However, Indirect I/O requests are tougher to account to. Additional information must be extracted from the request to determine which tenant the request belongs to, as it does not come directly from the tenant.

4.1.3 Governing Multiple Logical Drives

A logical drive is a collection of disk spindles that is exposed to the OS as a single device. A file on the drive is generally striped across all spindles. Most DBMSs use multiple disks, with data striped across almost all the disks, and one allocated as the controller. SQLVM governs each disk independently, so an IOPS reservation for a tenant maps correctly.

4.2 I/O Metering

Narasayya describes the I/O metering logic by using a running example. With a reservation for 100 IOPS for a tenant, the promise can have two different interpretations. The strong version operates such that as long as the tenant has at least one I/O request pending then the system will issue one I/O every 10 msec. For this, the metering logic computes the delay d between when the I/O should have been issued and when it is actually issued. The scheduling mechanism maintains a deadline for each I/O that identifies the time by which the I/O must be issued to achieve the desired $ResIOPS$. If the delay is 0 for every I/O request, then $ResIOPS$ has been met.

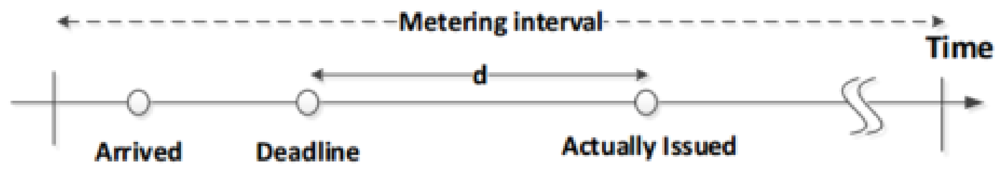


Figure 3. Image source: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

The weak version promises that the average IOPS is at least 100 IOPS. This version relaxes the requirement to issue one I/O every $\frac{1}{ResIOPS}$ sec as long as the average rate for the interval meets the promised $ResIOPS$. Metering for this also leverages the deadline mechanic. At the end of the interval, each I/O request whose deadline lies within the interval but was not issued is a violate. If there are *n* such violations in a metering interval of *t* sec, then the promise is violated by $\frac{n}{t}$ IOPS (Narasayya, 2013).

5. Experiments

Narasayya and his team used four different workloads in their tests: TPC-C, Dell DVD Store benchmark, TPC-H, and CPUIO. Both the TPC-C and Dell DVD Store workloads are Online Transaction Processing (OLTP) style workloads. The TPC-H workload is a DSS workload, meaning it simulates batch processing. Finally, the CPUIO workload is designed to generate 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 are designed to mimic a supplier's standard data usage, including a good mix of read/write transactions where more than 90% of said transactions include at least one write operation.

The Dell DVD Store benchmark represents an e-commerce workload, simulating standard user interactions with a database through a web application.

The TPC-H benchmark simulates twenty-two business-oriented ad-hoc queries. This simulates business systems that manage large amounts of data, execute complex queries, and supply answers to business questions.

The CPUIO benchmark has a single table, with a clustered index on the primary key and a non-clustered index on the secondary key. This benchmark consists of three query types: a CPU-intensive computation, a query involving a sequential scan with a range predicate on the primary key of the table, and a query with a predicate on the non-clustered index which performs random accesses on the database pages.

During the experiments, each tenant connects to a separate database and executes an instance of one of these workloads. The tenants were hosted within a single instance of the database server, with a 12-core processor, three striped Hard Disk Drives (HDD), the transaction log stored on a Solid State Drive (SSD), and 72 GB of memory.

5.1 Meeting Reservations

This experiment uses a micro-benchmark to evaluate SQLVM's ability to meet the resource reservations when enough resources are available at the database server. First, the focus is on IOPS and CPU Utilization. The next test focuses on one resource at a time to rule out interactions between resources as an impact. Afterwards, the focus is on generating CPU- and I/O-intensive workloads. Finally, two tenants are co-located, executing identical workloads but with different resource reservations.

The first test focuses on IOPS. The CPUIO benchmark only issues I/O-intensive queries with minimal CPU requirements. Tenant T_1's SQLVM was set to $ResIOPS_1 = 60$, and T_2's configuration is $ResIOPS_2 = 20$.

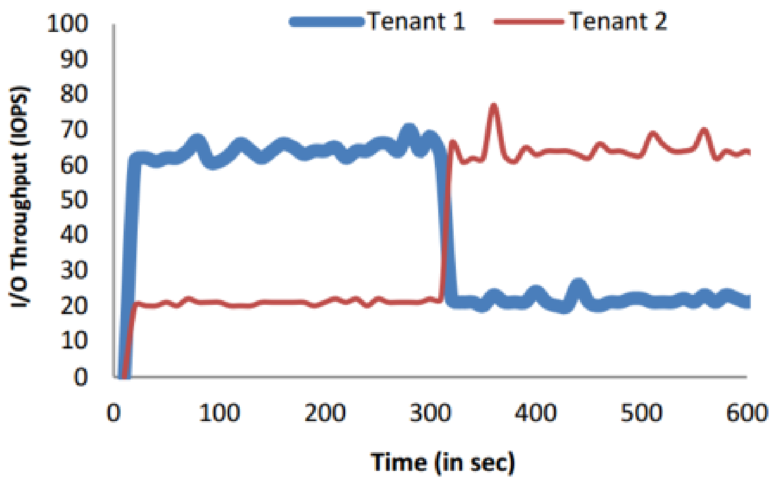
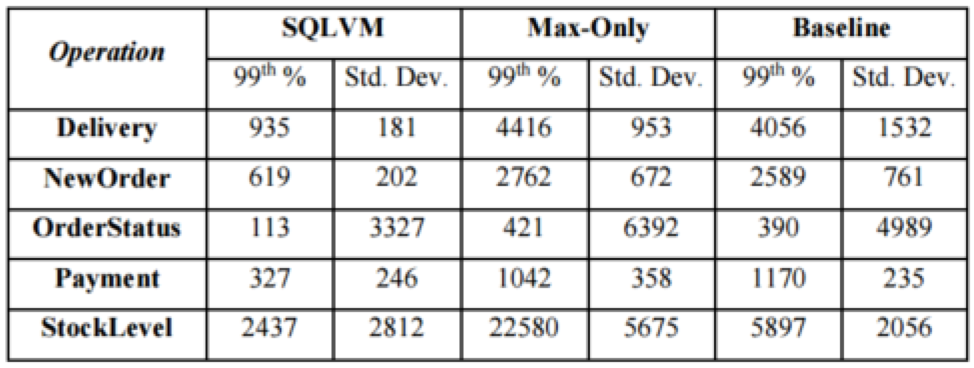


Figure 4. Image source: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

5.2 Performance Isolation

In the performance isolation experiment, four tenants executed concurrently one instance of each workload. As per their tests, the resource isolation in SQLVM resulted in significantly better performance isolation than the alternative configurations. In a further experiment with twenty-four other tenants, T_1 (running in SQLVM) had up to six times the throughput. The following table outlines T_1's performance against the other tenants.

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

Image source: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

5.3 Validating Metering

The previous experiments focused on scenarios where the server had sufficient resources to meet the promised reservations. However, this does not sufficiently test SQLVM's metering mechanics. In this experiment, eight tenants are co-located on the same SQL Server instance. Each tenant is promised $ResIOPS_i = 80$. The aggregate of all I/O reservations is 640 IOPS, which is more than double the approximate 300 IOPS capacity of the disks.

Figure 5 shows two of the eight tenants, performing I/O requests. T_1 is executing TPC-C and T_2 is executing the Dell DBD Store workload. 300 seconds into the experiment, when the fourth tenant starts executing, the IOPS achieved by both tenants is lower than their reservations. Therefore, the promises might be violated, assuming they had pending I/O requests.

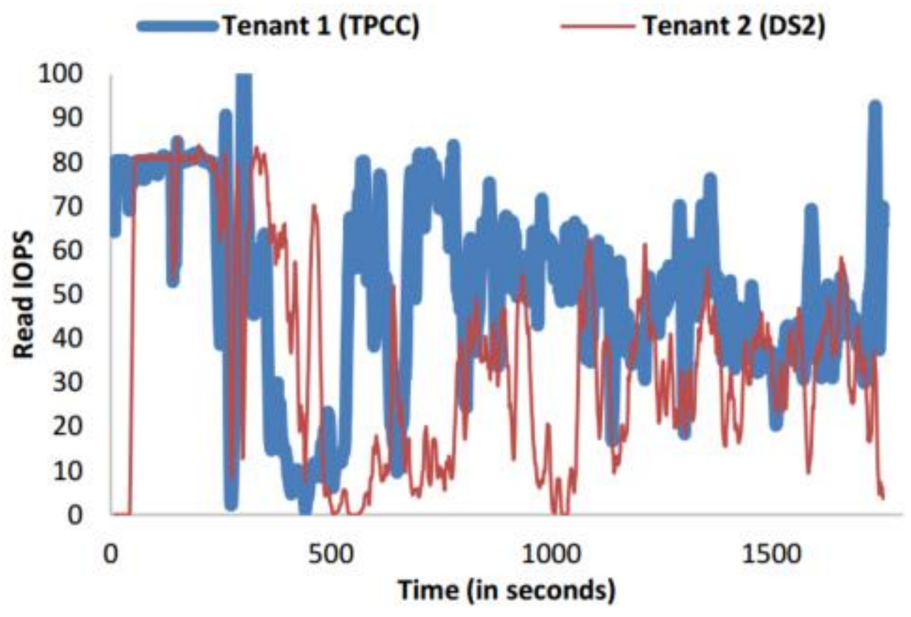


Figure 5. Image source: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

Figure 6 plots the I/O violations for the two tenants. The y-axis plots the number of I/O operations that were tagged to be issued in a metering interval, but were not due to insufficient capacity.

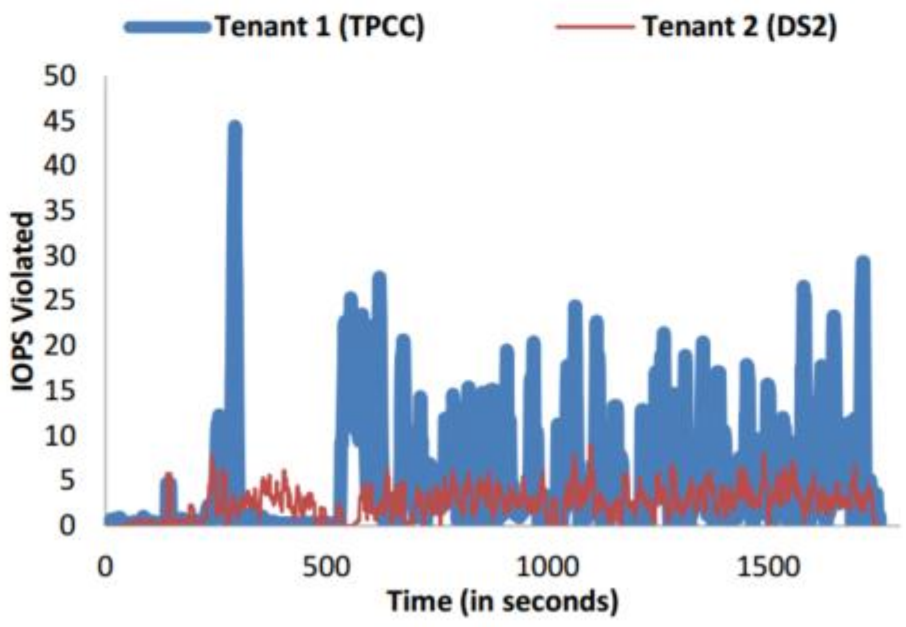


Figure 6. Image source: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

The metering logic's ability to differentiate between a violate and insufficient requests is shown in Figure 6.

6. Critique

This paper does a very good job of explaining their reasoning and mathematical choices. The authors go into sufficient detail explaining how the SQLVM promises and metering will improve the throughput of multi-tenant systems.

Another strength of the paper is their related work section. The authors do a very good job of comparing their work to other work being done by other groups. They also go into detail to explain how various related works integrate and strengthen the SQLVM proposal. This was done very well and in great detail, and goes a long way to helping validate their claims and data.

The paper's main weakness stems from the assumption that you already understand the concepts involved in multi-tenant systems. This required extra research to make sense of the original paper. An explanation of multi-tenancy is provided in section 2 here, and helps explain why the authors and researches took this approach to managing resources. Understanding multi-tenancy is critical to understanding the concepts and reasoning behind the SQLVM architecture.

Another weakness of the paper, though an understandable one, was the lack of details on how the authors actually implemented the SQLVM on SQL Azure. However, this is understandable given that they are Microsoft employees, working in the Azure group, with higher access and understanding of Azure. Unfortunately, this makes it more difficult to conduct our own experiments to verify the team’s findings.

7. Conclusion

In conclusion, the SQLVM paper does a very good job of proposing the SQLVM architecture and explaining the reasons behind each architectural decision. The SQLVM promises a specific threshold of resources to each tenant in a multi-tenant Database-as-a-Service (DBaaS) system, and meters the results to ensure tenants do not violate their resource allocation. This is critical for reducing overhead in large DBaaS offering. The overhead reduction leads to large cost savings, which is both very desirable for any company, and also makes it easier to create larger and more powerful systems. The experiments the authors ran showed significant throughput improvements when tenants used the SQLVM prototype they developed in SQL Azure.

The paper had many strong points, including very detailed explanations of the math involved in managing the resources in a multi-tenant system. The authors explained every step and decision in a sufficient level of detail to both understand their proposal, and to be able to support the results. This critique explored multi-tenancy in more detail, to fill in any knowledge gaps in the original paper, however the paper makes a very strong argument to those with an understanding of multi-tenancy.

Citations

Betts, D. (2012). Developing multi-tenant applications for the cloud on Windows Azure. Redmond, WA: Microsoft.

Chong, F., Carraro, G., & Wolter, R. (2006, June). Multi-Tenant Data Architecture. Retrieved March 30, 2016, from https://msdn.microsoft.com/en-us/library/aa479086.aspx

Gollnick, J. (2016, March 25). Choose a cloud SQL Server option: Azure SQL (PaaS) Database or SQL Server on Azure VMs (IaaS). Retrieved March 30, 2016, from https://azure.microsoft.com/en-us/documentation/articles/data-management-azure-sql-database-and-sql-server-iaas/

Lang, W., Shankar, S., Patel, J. M., & Kalhan, A. (2012). Towards Multi-Tenant Performance SLOs. *2012 IEEE 28th International Conference on Data Engineering,* 702-713.

Narasayya, V., Das, S., Syamala, M., Chaudhuri, S., Li, F., & Park, H. (2013). A demonstration of SQLVM. *Proceedings of the 2013 International Conference on Management of Data - SIGMOD '13*. Retrieved March 30, 2016, from http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

Narasayya, V., Menache, I., Singh, M., Li, F., Syamala, M., & Chaudhuri, S. (2015). Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow. Proceedings of the VLDB Endowment, 8*(7), 726-737. doi:10.14778/2752939.2752942

Urgaonkar, B., Shenoy, P., & Roscoe, T. (2002). Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review SIGOPS Oper. Syst. Rev., 36*(SI), 239. Retrieved March 30, 2016.