Scaling Off-the-Shelf Databases with Vela: An approach based on Virtualization and Replication

Tudor-Ioan Salomie *
Google Inc.
tsalomie@google.com

Gustavo Alonso
Systems Group,
Computer Science Department,
ETH Zürich, Switzerland
alonso@inf.ethz.ch

Abstract

Off-the-Shelf (OTS), relational databases can be challenging when deployed in the cloud. Given their architectures, limitations arise regarding performance (because of virtualization), scalability (because of multi-tenancy), and elasticity (because existing engines cannot easily take advantage of the application migration possibilities of virtualized environments). As a result, many database engines tailored to cloud computing differ from conventional engines in functionality, consistency levels, support for queries or transactions, and even interfaces.

Efficiently supporting Off-the-Shelf databases in the cloud would allow to port entire application stacks without rewriting them. In this paper we present a system that combines snapshot isolation (SI) replication with virtualization to provide a flexible solution for running unmodified databases in the cloud while taking advantage of the opportunities cloud architectures provide. Unlike replication-only solutions, our system works well both within larger servers and across clusters. Unlike virtualization only solutions, our system provides better performance and more flexibility in the deployment.

1 Introduction

When deploying applications on the cloud there is a trade-off between using Off-the-Shelf (OTS) databases or cloud-ready solutions. While using OTS databases is convenient as entire application stacks can be migrated to the cloud, there is no efficient way of doing this yet [1]. Ad-hoc solutions that try to virtualize OTS databases often hit performance bottlenecks due to virtualization [6] or a miss-match between what virtualization offers and what OTS databases support [34]. Recent trends in increasing number of cores in cluster nodes also add to the complexity of the problem. OTS databases and replication-based solutions built on top of them have difficulties in harnessing these computational resources [19, 29, 7].

Cloud-ready solutions offer scalability and elasticity at the cost of reduced functionality, varied levels of consistency, or limited support in queries and transactions. For instance, Dynamo [13] sacrifices consistency

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*The work presented in this paper was done during the author's PhD studies in the Systems Group at ETH Zürich, Switzerland and is in no way affiliated, nor does it represent the views of Google Inc.

under certain scenarios and has a simple put/get query API. Bigtable [10] also has a simple query API that does not support a full relational model.

Relational systems like SQL Azure [9] use replication for fault tolerance and load balancing, but the replication factor is fixed with no way of tuning it in favor of fault tolerance or performance. Similarly, Amazon RDS [2] relies on replication but imposes a variety of restrictions based on the chosen underlying database engine.

In this paper we present **Vela**, a system for running OTS databases on the cloud. The system relies on *virtualization* (to take advantage of the system architectural possibilities available in the cloud) and snapshot isolation (SI) *replication* (for fault tolerance, performance, and scalability, without sacrificing consistency). Synchronous replication has become a de facto standard for achieving scalability and availability with industry solutions like SQL Azure [8], Megastore [3], or Oracle RAC [25] relying on it.

Novel virtualization related techniques such as memory ballooning [4], hot plugging/removing cores or live-migration [11] add flexibility and support for online reconfiguration to Vela.

Although based on known techniques like replication and virtualization, the design of Vela faced several non-trivial challenges. The biggest ones were in designing our system such that it can seamlessly take advantage of both the size and number of cluster machines over which it is deployed. First, handling the two dimensions (of size and number) separately makes reasoning and taking decisions (e.g., deployment, online reconfiguration) about the system more difficult. Virtualizing the hardware reduces the complexity with a minimal cost in resources and performance. Further, having a homogeneous view of the machines allowed us to define an API for the online reconfiguration of the system. Second, managing a dynamic set of replicated OTS database instances over a virtualized cluster required a coordinator component that implements the replication logic, monitors the deployment, and automatically takes decisions in order to react to load changes. Third, addressing performance and functional miss-matches between databases and virtual machine monitors required engineering effort in order to achieve good scalability and support the reconfiguration API.

The performance evaluation of the system, relying on standard workloads (e.g., TPC-W and TPC-E), shows that it scales both on large servers and on clusters, it supports multi-tenancy and can be dynamically reconfigured at run-time. In scalability tests, spanning two dozen nodes, the system handles more than 15k transactions per second servicing transactions for 400 clients, while maintaining response times less than 50ms. The multi-tenancy experiments show that the system can easily accommodate 20 collocated tenants while maintaining response times below 400ms under an aggregated load generated by 800 clients.

The main contributions of the paper are the following:

- To the best of our knowledge, Vela is the first solution that treats clusters and multicore machines as a uniform pool of resources for deploying and managing a replicated database system, an architectural step necessary to operate in modern cloud systems.
- Vela demonstrates how to combine virtualization and replication such that the benefits of both can be simultaneously achieved: a scalable system that does not have a static deployment and can be reconfigured at runtime.
- The evaluation of the system illustrates many important technical design choices that lead to a system that is generic (handling different read- intensive workloads), scalable, online reconfigurable and supports multitenancy.

2 Motivation

The main argument in favor of using Off-the-Shelf (OTS) databases in cloud setups comes from the shortcomings of existing cloud-ready solutions. Existing solutions like Dynamo [13], Dynamo DB or Bigtable [10] require the applications running on top to implement logic that traditionally resides in the database. This includes consistency guarantees, support for transactions over the whole data-sets, or support for complex queries and operators (e.g., join).

There are also counter arguments for running OTS databases in the cloud. Virtualization, elasticity, and online reconfigurability are common traits of cloud systems which do not match the design of OTS databases as these are usually deployed on dedicated servers and statically provisioned with resources (CPU, memory) for peak loads.

In virtualized environments, hot plugging / removing of resources is a common mechanism for online reconfiguration, supported by most hypervisors like Xen or VMWare. While OTS databases, both commercial (SystemX) and open source (PostgreSQL, MySQL) will react to hot plugging or removing of CPUs, they face larger challenges with memory. This comes from the fact that databases take memory management under their own control (to different degrees). For example, commercial SystemX is statically configured with a fixed memory size that serves as both data-cache and as a memory pool used for sorting or aggregation operations. At the other extreme we find PostgreSQL which liberally only takes a small amount of memory (i.e., the "shared_buffer", recommended $\frac{RAM}{4}$) under its control, relying on the OS disk cache for data-caching.

For OTS systems like PostgreSQL, dynamic memory plugging or removing can be used out of the box, while for systems like InnoDB or SystemX, Application Level Ballooning [22, 31] mechanisms are needed.

While OTS databases provide functionality lacking in readily available cloud solutions, a naïve solution does not work out of the box. We show however that with the design of Vela, elasticity, collocation and online reconfiguration can be achieved.

2.1 Replication is not enough

Snapshot Isolation based replication is commonly used in both commercial products (SQL Azure [8] built on top of Cloud SQL Server [5] or Teradata [40]) and research prototypes ([21], Tashkent [14], Ganymed [28]) as a means of scaling read intensive workloads over compute clusters and to increase availability.

Where these systems come short is user controlled dynamic reconfiguration or flexibility. For instance SQL Azure [8] does not emphasize virtualization or flexibility in the replication configuration.

Its Intra-Stamp replication offers fault tolerance and load balancing by having a primary master and two secondary replicas, synchronously replicated. The Inter-Stamp asynchronous replication is used for geo-replication and disaster recovery, as well as a mechanism for data migration.

The authors of [16] investigate the performance of SQL Azure as a black box. The points of interest are the network interconnect inside the cluster (peaking 95% of the time at 90MB/sec) and the cost of a TCP RTT, which is below 30ms for 90% of the samples. Also, under a TPC-E [36] like workload, they observe that the system scales up to 64 clients. In our evaluation we show that Vela scales well beyond this in an environment with a similar network configuration.

While these systems scale their replicas over clusters, they rely on the database engine's parallelization to scale up with the individual resources of each cluster node. As servers have increasingly more cores, many new factors impede traditional relational databases from scaling up. Some of them include contention on synchronization primitives in a database [18, 19], workload interaction in the presence of many real hardware contexts [32] as well as the effects of hardware islands in database deployments on NUMA systems [29].

Scaling up refers to the ability of the system to usefully utilize the resources of a single machine and to yield performance gains as more resources (in our case CPU and memory) are added. Scaling out refers to the ability to utilize and gain performance benefits from adding more machines (cluster nodes) to the system. In most replication based systems, scaling out and up are seen as two orthogonal problems as there is no uniform view of resources within a machine and in the cluster. Using virtualization over clusters of multicore machines, it is possible to define a uniform resource pool, with replication used for scaling both up and out.

2.2 Virtualization is not enough

Favorable arguments for running databases in virtualized environments emphasize consolidation opportunities and dynamic reconfiguration of allocated resources.

Recent work shows a large adoption of virtualized databases for multi-tenancy. Tuning database performance by controlling the encapsulating virtual machine [34] or optimal data placement in the storage layer based on the tenant workload characteristics [26] are commonly investigated.

Virtualizing OTS databases may lead to a degradation in performance. A few studies have investigated this [6], corroborating our experience in building Vela. In most cases, main memory workloads behave very similarly in Bare Metal vs. Virtualized setups, mostly due to Intel's VT-x and AMD's AMD-V support for virtulization. For I/O intensive workloads, virtualized performance degrades and CPU utilization increases, as compared to Bare Metal.

Counter arguments for virtualizing databases hint at resource overheads. The Relational Cloud [12] system makes the case for a database-as-a-service that efficiently handles multi-tenancy, has elastic scalability and supports privacy. The authors argue that achieving multi-tenancy by having a database in a VM architecture is inefficient due to additional instances of the OS and database binaries. In our evaluation of Vela, we have not seen the memory consumption overhead of the OS and database binaries to be an issue, being far lower than that of the data being processed.

We consider virtualization to be a suitable solution being a proved technology that is readily available and that requires no modifications to the existing database engines. It allows fast deployments, easy manageability, implicit resource separation, and offers means for online reconfiguration of the system through techniques like live migration, memory ballooning or hot plugging of CPU cores.

2.3 Dynamic reconfiguration

Dynamic system reconfiguration addresses the system's ability to change (at runtime) at different levels of granularity. Solutions like SCADS [37] or that of Lim et al. [20] do automatic scaling only through coarse grained operations by adding / removing physical servers. DeepDive [24] reconfigures the collocation of VMs in case of performance interference. These solutions do not cover fine grained reconfiguration like hot plugging / removing of memory or cores in VMs. Other approaches study finer grained online reconfiguration. For instance Microsoft SQL Server's dynamic memory configured on top of Hyper-V [30] and "Application Level Ballooing" as a generic memory ballooning mechanism implemented for MySQL [31] demonstrate that memory can be dynamically added or removed from databases running in VMs. However, neither takes into account coarse grained operations.

Complementing mechanisms for online reconfiguration, research looks at policies that drive them. For example, dynamic resource allocation for database servers for proportioning caches and storage bandwidth from a virtual storage server is described in [35].

Besides virtualization, there are also custom resource management and task scheduling solutions for large data-centers. Projects like Omega [33] or Mesos [17] are generic frameworks for running varied jobs in data-centers. They are not designed for relational data processing and might require re-engineering the database engines in order to be suitable for running on top of them. Both projects avoid virtualization for the purpose of managing the global set of resources in a data center, arguing that seemingly small resource overheads can be huge at scale.

In contrast to most of the existing cloud database systems for OTS databases, Vela emphasizes scaling both up and out and has an API for online reconfiguration. Instead of relying on data replication for durability, it uses replication for improving latency and scalability, decoupling the update workload from the read-only workload. Vela expands on using virtualization for supporting online reconfiguration with minimal overhead in resource utilization and performance.

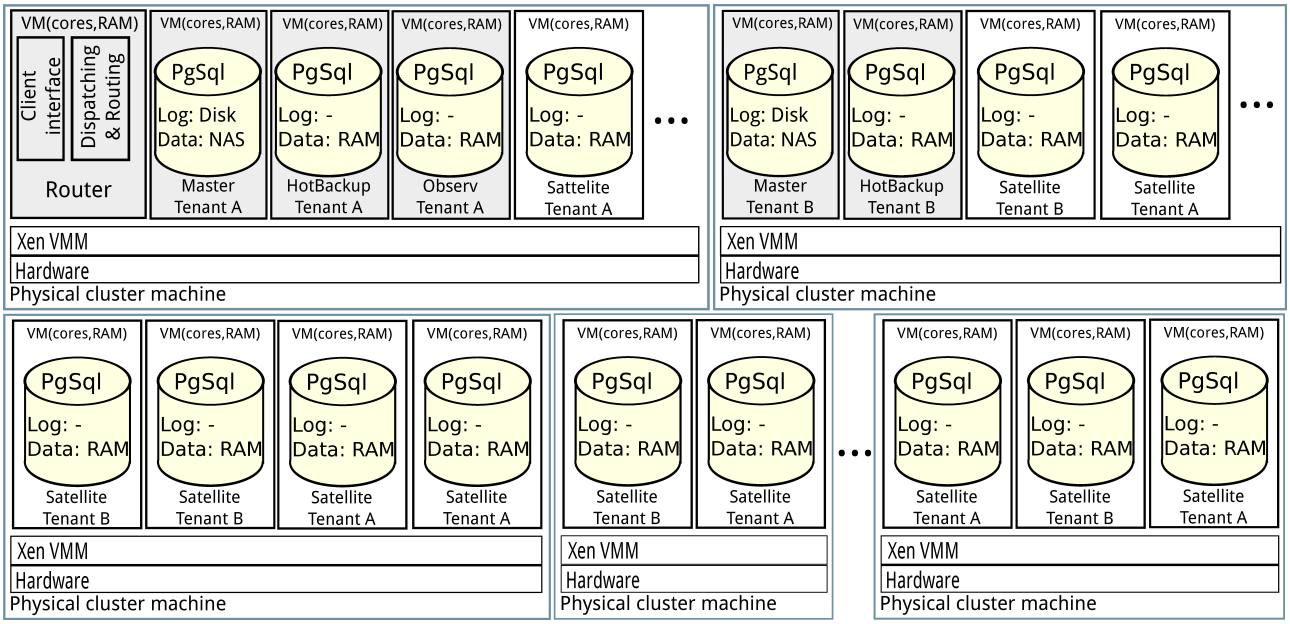


Figure 1: Architecture overview: an example deployment over a cluster of multicores

3 System architecture

Vela achieves scalability both within the boundaries of the same multicore machine and in a cluster of multicores by relying on single master data replication, similar to the Intra-Stamp replication of WAS [8] or Ganymed [27].

In this replication model, we define a "master" copy of the data and a series of replicas ("satellites"). All update transactions go to the master while read transactions go to the satellites, which are kept in sync with the master. Upon committing an update transaction, the master's server count number (SCN) is incremented and its change set is extracted and propagated to all satellites along with the SCN. Satellites increment their SCN after applying the changes from the master in the same order. Each transaction is tagged with the current SCN of the master when entering the system.

The system guarantees snapshot isolation as its consistency level. When using snapshot isolation the queries are guaranteed to see all changes that have been committed at the time the transaction they belong to started (a form of multiversion concurrency control that is present in many database engines such as Oracle, SQLServer, or PostgreSQL).

3.1 Replication

Vela extends the design of Multimed [32], a similar system designed for scaling transactional data processing on large multicores, though lacking support for scaling out over clusters, virtualization and dynamic reconfiguration. Figure 1 illustrates the components of the system, in a possible deployment over a cluster of multicores. At a high level Vela is composed of different Data Processing Instances (DPIs) and a Router.

The **Router** accepts requests and returns responses to and from the clients.

Worker threads dispatch requests to the databases in the Data Processing Instances (DPIs). The Router also extracts the changes to the data on the Master (called WriteSets) and propagates them to the other DPIs in the system. Each DPI is backed up by a database engine holding a copy of the data and is allocated a dynamic set of resources (CPU, memory). Each database within a DPI holds two data components: the actual data and the transaction logs. The options for storing the two can be specified for each replica as: RAM, local disk, or remote storage (NAS).

Like Amazon RDS or SQL Azure, our system uses existing relational database engines in its design. We chose PostgreSQL as the supporting database as it is open-source, with a well understood behavior and shows good standalone scalability on multicore systems. Also, PostgreSQL by its data cache design can react out of the box to memory ballooning operations in VMs. Replacing it with another database engine is straightforward as the only engine specific part is the WriteSet extraction mechanism. This is well described in Multimed [32].

The DPIs in Vela can have different types: Master, Satellite, HotBackup, or Observer. As shown in Figure 1, the Router and each DPI is encapsulated in a virtual machine. Vela supports multi-tenancy by allowing multiple Masters. Each Master DPI has its own set of Satellite, HotBackup and Observer DPIs. In the configuration shown in Figure 1, Vela handles two tenants (A and B), each having a Master, a HotBackup and multiple Satellites. Tenant A also has an Observer DPI.

The **Masters** are primary copies of the data and are configured to offer durability, with the transaction log on a local disk and the data on either local disk or NAS. Alternatively, in a non-durable setup, the Master can be entirely held in RAM. The Router dispatches each tenant's update transactions to its Master.

Satellites hold replicas of the data of a Master. The Router assigns the read-only transactions to Satellites that have the SCN matching the transaction's SCN. When more Satellites fulfill this condition, the one with least load is chosen. The load is defined as the number of current active transactions bound to a Satellite. Satellites do not need to ensure durability (but could be configured to do so).

The **HotBackup** and **Observer** are special and optional types of DPI. HotBackups do not process client requests, though they are kept up to date, receiving all the WriteSets. HotBackups are used for spawning new Satellites at runtime for each tenant. Observers are used to approximate the best response time for the read-only transactions of a tenant. Vela ensures that there is always one read-only transaction routed to each Observer.

The Satellites, HotBackups and Observers do not need to offer durability. As most database engines do not have an option for disabling the transaction log, we emulate this by storing it in RAM and disabling synchronous commits. The overall system durability is given by the Master. While the transaction logs are small enough to be always held in RAM, the actual data might be too large for this. In this case, the data is held on NAS and the available RAM is used for caching. The reconfiguration API supports adjusting the memory of each DPI for increasing or decreasing the memory cache. As PostgreSQL relies on the disk cache for most of its data caching it requires no modifications to be able to adapt to changes in the DPI's memory. For database engines that do their own memory management, a virtulization based solution like Application Level Ballooning [31] can be used.

3.2 Virtualization

In Vela, all the DPIs are spread out over the available resources in a cluster of multicores. We have looked at different options for managing the resources (CPU, RAM) of a cluster of multicores and formulate Vela's requirements for a resource management system:

- *Multiplexing* the hardware for running multiple Data Processing Instances on the same physical machine while allowing for NUMA-awareness. This enables Vela to scale up data processing.
- *Isolation* of CPU and memory allocated to each Data Processing Instance, such that it can make assumptions on the current available cores and memory.
- *Reconfiguration* at runtime of the resources allocated to Data Processing Instances.
- *Uniform* view of the whole cluster. Migrating a Data Processing Instance from one cluster node to another should be seamless and with no service interruption.

In Vela we opted for virtualization (Xen Hypervisor [4, 39]), as it fulfills our requirements. The guest domains (DomUs) run in paravirtualized mode. We have observed no impact on the performance of the Satellites that have the dataset in main-memory. For the Master and the Router that are I/O intensive, the DomUs in their default configuration have quickly become the bottleneck. Similar observations [23] indicate that network I/O

can become a bottleneck very fast. *Latency* wise, we were able to tune the system through Linux Kernel network specific settings in both Dom0 and DomU. *CPU* wise, the problem is more involved. For network I/O, Xen allocates in the host domain (Dom0) one kernel thread for each virtual network interface (VIF) in the DomUs. DomUs hosting Routers that do high frequency network I/O were provisioned with up to 8 VIFs. In order to remove the CPU bottleneck of the VIF backing threads in Dom0 we allocated a number of cores equal to the total number of VIFs of all DomUs. Alas, the *interrupts* generated in Dom0 and DomUs need to be balanced over all available cores. Failing to do this leads to one core handling most IRQ requests, becoming a bottleneck.

3.3 Dynamic reconfiguration

Databases deployed on one server are traditionally given exclusive access to the machine's resources. They are generally also over-provisioned to handle peak load scenarios. Vela tries to avoid this by dynamically adjusting the resources of each DPI and the number of DPIs in the system to the load.

We differentiate between two types of reconfiguration operations: DPI level (fine grained) reconfiguration and System level (coarse grained) reconfiguration. At the DPI level, we can reconfigure the provisioned number of cores and the amount of memory. At the System level, we can reconfigure the number of DPI as well as their placement in the cluster. **addCoresToInstance** and **removeCoresFromInstance** control the number of cores allocated to each DPI. They rely on Xen's ability to hot plug cores in a VM. **increaseInstanceMemory** and **decreaseInstanceMemory** control the amount of RAM allocated to each DPI. Increasing / decreasing a DPI's memory uses Xen's memory ballooning. **addInstance** operation is used to spawn new DPIs. A new Master is added to the system for each new tenant, while a new Satellite can be added to mitigate an increase in the load of a tenant. Adding a new Master requires spawning a new VM and starting the Master database in it. Adding a Satellite is more involved. Besides spawning a new VM and starting a new database in this VM, Vela needs to ensure that the database is identical to the one in the Master. Instead of halting requests to the Master database and cloning it, Vela relies on the HotBackup described below. When removing a DPI (**removeInstance**), no more requests are accepted by the Master of the corresponding tenant. When all outstanding requests are served, the Master and all its Satellites are stopped (database then VM). The operation corresponds to removing a tenant from the system. In the case of stopping a Satellite, no more transactions are routed to it and once the outstanding ones complete, the database and the encapsulating VM are stopped and their resources (CPU and RAM) are freed. **moveInstance** operation handles moving a DPI in the cluster from one physical machine to another physical machine. Two types of "move" operations can be performed. A *cold* move will make the DPI unusable during the operation. This is implemented in Vela either by *removeInstance* followed by *addInstance* or through Xen's cold migration. With *hot* moving, the DPI is migrated while it is still serving request, relying on Xen's live migration. *Cold* moves are faster then *hot* moves. As Satellites in Vela do not have any state associated, except for the currently running transactions, they can always be *cold* moved. Masters however can never be taken offline and consequently are always *hot* moved.

The HotBackups enable adding new Satellites without taking the whole system offline. The overhead is minimal in terms of resources: the HotBackup requires only 1 core and main memory for the transaction log. The time it takes to copy the HotBackup into a new Satellite is bound by the network bandwidth. This time only influences the amount of main-memory needed by the queue that temporarily stores pending WriteSets for the HotBackup and new Satellite. The maximum size of the queue can be approximated: $\frac{Tx}{sec} \times \frac{avg(WriteSetSize)}{Tx} \times CopyTime(sec)$. For a tenant processing 10k transactions per second, with average WriteSets of 1KB per transaction (way more than the TPC-W average), deploying a 20GB dataset over a 1GBit network would grow the queue to a maximum of $\approx 1.5GB$ – which is easily manageable.

3.4 Automatic runtime reconfiguration

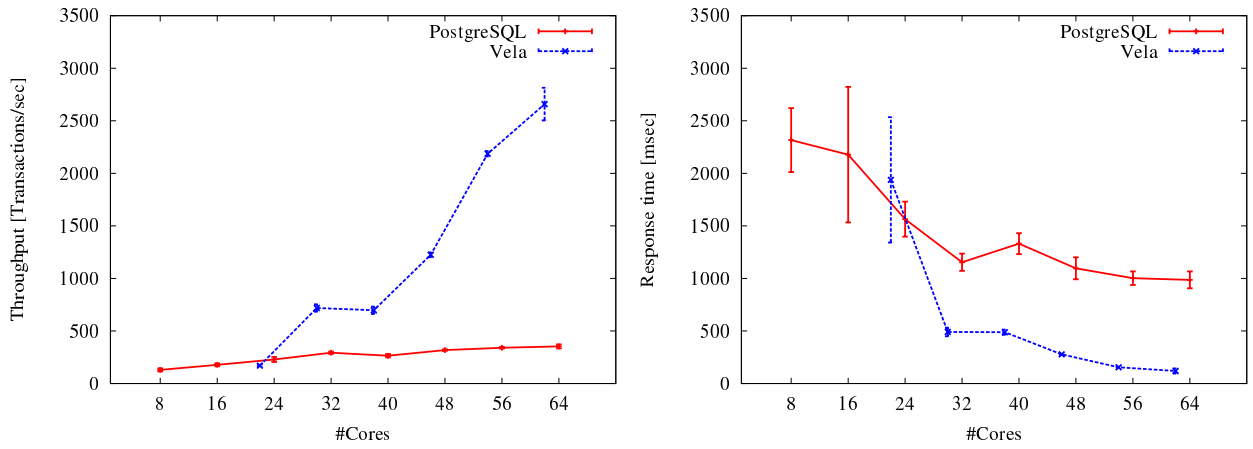
The dynamic reconfiguration of Vela can be done manually or automatically based on monitored metrics. The system reports low level metrics for each VM corresponding to a DPI (e.g., CPU utilizations, memory consumption and disk/network utilizations) as well as per-tenant statistics (e.g., overall throughput, read-only and update transaction latencies, etc.). The system maintains a set of user specified *target functions* that describe threshold values for monitored metrics and simple policies based on the reconfiguration API for achieving these thresholds. Basic tenant SLAs, like desired response time, can be expressed through target functions. The evaluation Section 4.2 exemplifies how the reconfiguration API and target functions work together.

For automatic reconfiguration support for target functions involving system response time, we rely on the Observer for determining the best response time of read-only workload. The Observer is an optional DPI, always deployed on 1 core. The Router ensures that if an Observer is present, it will always have one and only one transaction routed to it. As only one transaction is being executed at a time there is no contention in the Observer. Also as it runs on only 1 core it exhibits no scalability issues. The Observer approximates the best transaction latency for the current workload and reports it as a metric that can be used in target functions.

4 Experimental evaluation

This section presents the experimental evaluation of Vela, focusing on the three key aspects of the system: scalability, automatic online reconfiguration and support for multi-tenancy.

For the scalability study of Vela on large multicore machines, we used a 64 core (4 sockets$\times$16 cores) AMD Opteron 6276 (2.3GHz) with 256GB DDR3 RAM (1333MHz). For the scalability study over multiple machines, we used a cluster of 10 servers, each with 16 cores (Intel Xeon L5520 2.27GHz) with Hyperthreading enabled, with 2 NUMA Nodes, and a total of 24GB RAM. As the database engine we opted for PostgreSQL 9.2, used both as a baseline in standalone experiments (in a Native setup) and as the underlying database for Vela (in a Virtualized setup). We chose 2 read-intensive workloads from the TPC-W and TPC-E [36] benchmarks: TPC-WB $\approx 95\%$ reads and TPC-E $\approx 85\%$ reads. Focusing on the data-processing system, we have omitted the application stack from the benchmark and connect the clients straight to the database engine. Due to similarity in results and space constraints, we only present the results for TPC-WB here.



(a) Throughput (b) Response Time

Figure 2: TPC-WB – Scaling up with number of cores: Vela vs. Standalone PostgreSQL

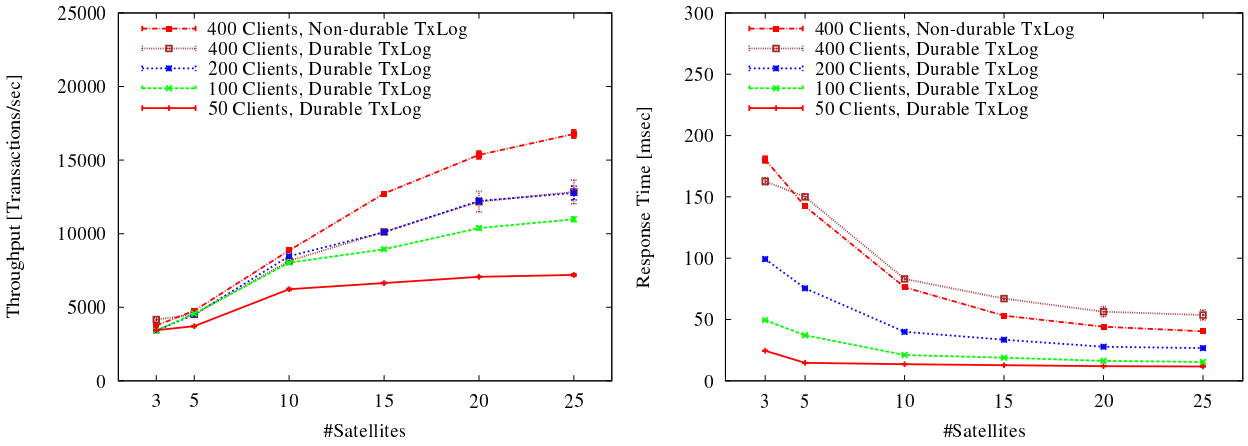
4.1 Scalability study

A main property of Vela is that it needs to scale with the resources it has been allocated. We present a series of experiments showing that Vela indeed seamlessly scales on multicores and on cluster of multicores, under different read-intensive workloads. Vela scales the read transactional workload until an I/O bottleneck is reached.

Scaling up

Figure 2 shows the scalability of Vela compared to that of Native PostgreSQL over a $\approx 23GB$ dataset. Both systems are under a load generated by 200 clients, each sending blocking requests, according to the TPC-WB benchmark. PostgreSQL has difficulties in scaling as number of cores it has at its disposal increases. Increasing by 8 cores at a time, the throughput grows slowly until 32 cores. Due to NUMA effects, after 32 cores, the performance drops, and slightly recovers as more cores are added. Vela's scalability line starts at 22 cores: 2 cores have been allocated to Xen's Dom0, 8 cores to Vela's Router and 12 cores to the Master. All subsequent datapoints correspond to adding additional Satellites, each assigned 8 cores. With the first Satellite, the throughput increases, as read load if offloaded to the Master (since the single Satellite has difficulties in keeping up with the incoming WriteSets). Once the second Satellite is added, performance does not increase as no more read transactions are being routed to the Master, and are handled by the two Satellites that can now keep up with the Master. With subsequent Satellites, throughput increases linearly up to the 5^{th} satellite, point at which the Master's CPU becomes the bottleneck.

It is clear that Vela has an added overhead of resources when deployed on a single machine. The cost of the Router, the Master and the Virtualization layer add up to the used cores and main-memory. Even so, Vela outperforms a traditional database by reducing contention on synchronization and minimizing workload interaction. Vela also exhibits a better scaling trend with the number of cores.



(a) Throughput (b) Response Time

Figure 3: TPC-WB – Scaling out over a cluster of multicores

Scaling out

Using the same architecture Vela also scales over clusters of multicores until either the Master's transaction log disk becomes the bottleneck or the network's bandwidth is fully utilized.

Figure 3 shows Vela's scalability as increasingly more Satellites are added, while blocking clients are issuing requests according to the TPC-WB benchmark. The dataset is $\approx 6GB$. All Satellites are configured to use 4 cores, and hold the data and the transaction logs in main memory. The Master is allocated 14 cores in a VM using up all the cores and memory of a cluster machine (16 total cores with 2 pinned to Dom0). The Master holds the data on NAS with the transaction logs on a local SSD drive, and has the shared buffers large enough to cache the data in main memory. The Router is running on a Native system (without virtualization) on an entire cluster node, in order to avoid the network issues described in Section 3.2.

As we increase the number of Satellites, we expect a linear increase in throughput. First we observe that for 50 and 100 clients, the system is underloaded. With 200 and 400 clients, the throughput is the same (lines completely overlap). At this point the transaction log disk of the Master is the bottleneck. Finally, giving up durability in order to further push the scalability of the system, we moved the Master's transaction log in main memory. With load offered by 400 clients, we see a further increase in throughput. The flattening of the throughput line from 20 to 25 satellites is this time due to reaching a CPU bottleneck on the Master, which can be solved by migrating it to a machine with more cores. At this point, the latency of the read workload cannot be improved and most of the 400 client connections end up being update transactions in the Master.

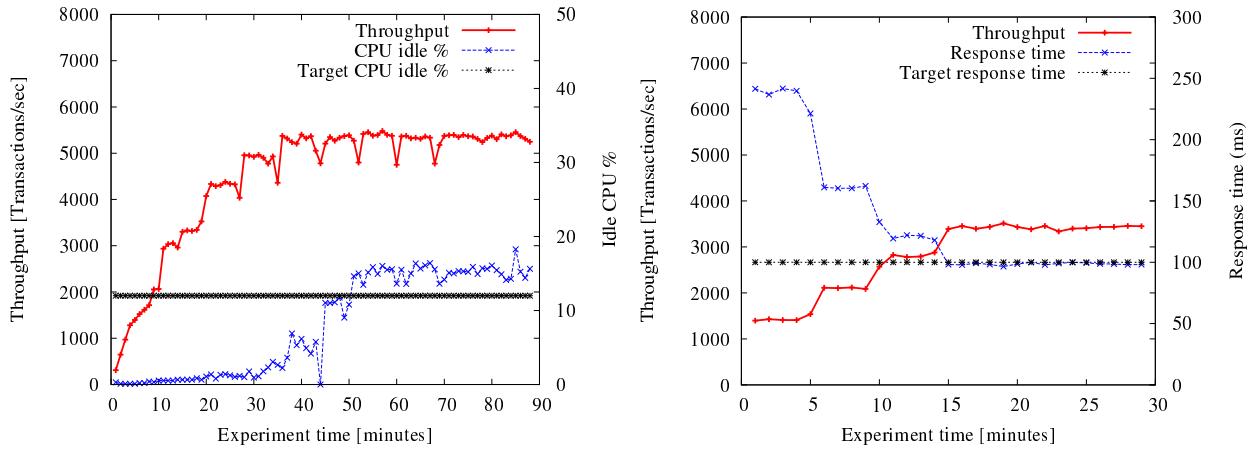
We point out that the flat throughput between 3 and 5 Satellites is caused by contention in the Satellites which cannot keep up with the Master. Consequently the Master also receives read-only transactions, improving the throughput. Scaling both up and out, Vela reaches high throughput rates over a commodity multicore cluster, processing more than 15'000 transactions/sec with latencies less than 50ms.

4.2 Elasticity study

All the experiments in the elasticity study were carried out over the same cluster used in the scale out experiments. This section presents both manual and automatic online reconfiguration of Vela, under a constant load of 200 clients issuing requests based on the TPC-WB benchmark. While the reconfiguration API offers support for shrinking the system, we present policies only for dynamically expanding the system.

Target functions

Manually monitoring and reconfiguring the system is an option. Alternatively, knowing what are the possible bottlenecks, they can be automatically addressed using target functions.



(a) Router core count (b) Satellite count

Figure 4: Automatic system reconfiguration

In Figure 4 (a) we show an experiment in which the system is under a heavy load based on TPC-WB. Knowing that the Router can become a bottleneck under heavy load and as it is CPU intensive, we set a target function on its CPU idle percentage.

Vela monitors the CPU idle % of the Router and uses the reconfiguration API to adjust the system. The target function monitors the CPU idle %. As long as it is less then 12% of the CPU cycles, the Router core count is increased. If no more cores are available on the current cluster node, the Router is live- migrated on a cluster node with more free cores. The reconfiguration stops when the target function is reached or when the live migration cannot be done. The plot shows the system throughput on the primary y-axis, CPU-idle percentage on the secondary y-axis and the experiment time on the x-axis. The throughput grows as more cores are added to the Router. Once the target function of 12% idle CPU is reached, the throughput is stable.

Target functions can monitor response variables that are specific to Vela, not only on generic ones like CPU or memory. Figure 4 (b) shows a target function set on the average response time of the system. The primary y-axis shows the throughput and the secondary y-axis the response time. The x-axis is the experiment time. A target value of 100ms for the average response time is set. The target function works by adding a new satellite (using the *addInstance* API) until the target response time is reached (the $check$ function).

The experiment starts with Vela configured with 1 Satellite of 4 cores. After 2 minutes a new Satellite is added to the system. Within less than 4 minutes the new Satellite receives load. This continues until the 4^{th} Satellite is added, point at which the response time falls under the target value (minute 15 onwards).

In this approach a cut-off value was supplied. This value is unfortunately workload dependent. Using the Observer described in Section 3.4, we monitor the best attainable response time for the current hardware, software & workload combination. The target for Vela can be set to a percentage of this value, making the system agnostic of the workload. We denote the optimal response time continuously measured in the Observer as $O(RT_t)$. Assuming that the Satellites and the workload are uniform, we probe the response time from a Satellite to get the runtime value $V(RT_t)$ at time t. The new $check$ function returns $V(RT_t) > \frac{100+tolerance}{100} \times O(RT_t)$, indicating if the overall system response time is matching the attainable response time, with the given $tolerance$.

```
testNextPackage()
1  if testIncrease(crtCores)
2     then crtCores++
3     else return FULL
4  if perfGain(crtCores)
5     then return OK
6     else crtCores--
7  return !OK

addSatellites()
1  while check(O,V) and haveHW()
2  do addSatelliteInstance()
3  if ! check(O,V)
4     then return DONE
5  return WARN(ProvisionHardware)

fillPackage()
1  while crtCores < coresInCrtPkg
2  do crtCores++
3  if ! check(O,V)
4     then return DONE
5  return !DONE

adapt()
1  if ! check(O,V)
2     then return DONE
3  repeat
4        if ! check(O,V)
5           then return DONE
6     until OK == testNextPackage()
7  return addSatellites()
```

Figure 5: Automatic satellite scaling pseudo-code

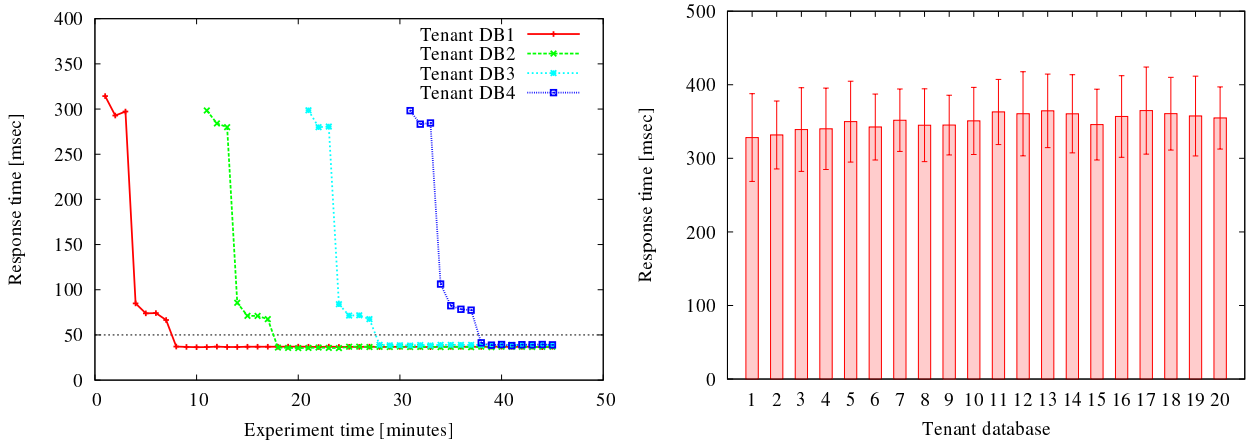
Automatic satellite scaling

Given a cluster of multicores we have shown that Vela can scale both up and out. One tradeoff has not be discussed so far: should Vela opt for few replicas with many cores (fat replicas) or for many replicas with few cores (thin replicas). Few fat replicas are easier to manage and require overall less RAM but will hit the scalability issues of individual databases. Many thin replicas increase the overall RAM consumption and transaction routing time. Each Satellite should be run at the optimal size with respect to the number of cores. Unfortunately the sweet spot depends on the hardware topology, database software and current workload. Based on empirical experience, we present an algorithm that adjusts the number of cores in a Satellite. It assumes that all Satellites are uniform and that the clients of each tenant have the same workload distribution. On the other hand it does tolerate changes in the workload, as long as all clients exhibit the same change.

From the experiments that we conducted, we observed that: (1) cache locality matters; and (2) cores that share the same cache behave similarly. The *adapt* function described in Figure 5 controls the process. Starting with each Satellite having 1 core, we expand it to all the cores sharing the same L2 cache (first package). If the required latency is not met, we expand to the cores sharing the LLC cache and then to different NUMA nodes. When moving from one level to another, we "probe" to see if a speedup is gained or not (*testNextPackage*). If no performance gain is obtained, then we stop increasing the cores. Otherwise we go ahead and allocate all the cores of that level to the Satellite (*fillPackage*). Once the size of a Satellite is determined, more Satellites of the same size are spawned (*addSatellites*) in an attempt to satisfy the $check$ function.

4.3 Multitenancy study

Scaling and dynamic reconfiguration in Vela work on a per-tenant base. This section shows that multiple Masters (each with its set of Satellites) can be configured under the same deployment of Vela. The dynamic reconfiguration plays an even larger role in this case. Satellites can be scaled for each tenant.



(a) Adapting to load

(b) High tenant concurrency

Figure 6: Multitenancy supporting auto-reconfiguration

Figure 6 (a) shows 4 tenants deployed in Vela. Each has its own Master and 1 Satellite with 1 core. Each tenant specified its target response time of 50ms. The plot shows the response time of each database tenant during the experiment time. The load is generated by 50 clients for each tenant and is based on the TPC-WB workload mix. The fast drop in response time captures the effect of increasing the Satellite cores from 1 to 4. This is insufficient to meet the latency target and Vela adds a new 4 core Satellite for the tenant (which takes about 5 minutes). At this point the target latency is met and the system stabilizes. Under similar load conditions the tenants behave similarly and in isolation. Moreover, Vela can use the set of cluster nodes that it manages to dynamically add Satellites for any of the tenants.

Figure 6 (b) shows Vela managing 20 tenant databases, each under a constant load from 40 clients (again, TPC-WB). All Masters are collocated on the same cluster node and each tenant has 1 Satellite. Within the standard deviation, all tenants perform similarly. The variation in response time is caused by contention on the transaction log storage of the cluster node holding the Masters.

5 Conclusions

We have presented Vela, a replication based system that scales Off-the-Shelf database engines both on large multicores as well as on clusters, using a primary master model. The system relies on virtualization for achieving a uniform view of all the available resources and also for supporting both fine and coarse grained online system reconfiguration. We also showed that multi-tenancy support is easily achieved by collocating database in the system.

Future directions for Vela include support for diverse Satellites optimized for certain transactions (e.g., heterogenous storage engines, partial replication, different indexes among replicas) or adding functionality to tenants (e.g., time-travel, sky-line or full text search operators) without affecting their performance. The applicability of specialized Satellites is not limited to performance and functional aspects. Vela could also be used for Byzantine fault tolerance, as proposed for heterogenous database replication systems [38, 15].

As shown in our evaluation, Vela handles diverse read-intensive workloads and exhibits little overhead from virtualization. Paying attention to engineering aspects, both in implementing and deploying the system, we

were able to achieve good scalability and high throughput rates while supporting transactional workloads with snapshot isolation guarantees.

References

[1] Ashraf Aboulnaga, Kenneth Salem, Ahmed A. Soror, Umar Farooq Minhas, Peter Kokosielis, and Sunil Kamath. Deploying Database Appliances in the Cloud. *IEEE Database Engineering Bulletin*, 32:13–20, 2009.

[2] Amazon RDS Multi-AZ Deployments `http://aws.amazon.com/rds/multi-az/`.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. of the 2011 Conference on Innovative Data system Research*, CIDR'11, pages 223–234, 2011.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM symposium on Operating systems principles*, SOSP'03, pages 164–177, New York, NY, USA, 2003. ACM.

[5] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for Cloud Computing. In *Proc. of the 27th IEEE International Conference on Data Engineering*, ICDE'11, pages 1255–1263, Washington, DC, USA, 2011. IEEE Computer Society.

[6] Sharada Bose, Priti Mishra, Priya Sethuraman, and Reza Taheri. Benchmarking Database Performance in a Virtual Environment. In *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 167–182. Springer Berlin Heidelberg, 2009.

[7] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, pages 143–157, New York, NY, USA, 2011. ACM.

[9] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme Scale with Full SQL Language Support in Microsoft SQL Azure. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 1021–1024, New York, NY, USA, 2010. ACM.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proc. of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[12] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database Service for the Cloud. In *5th Biennial Conference on Innovative Data Systems Research*, CIDR'11, Asilomar, CA, January 2011.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 205–220, New York, NY, USA, 2007. ACM.

[14] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-performance Scalable Database Replication. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys'06, pages 117–130, New York, NY, USA, 2006. ACM.

[15] Rui Garcia, Rodrigo Rodrigues, and Nuno M. Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *Proc. of the 6th ACM European Conference on Computer Systems*, EuroSys'11, pages 107–122, New York, NY, USA, 2011. ACM.

[16] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey. Early Observations on the Performance of Windows Azure. In *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC'10, pages 367–376, New York, NY, USA, 2010. ACM.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[18] Ryan Johnson, Manos Athanassoulis, Radu Stoica, and Anastasia Ailamaki. A New Look at the Roles of Spinning and Blocking. In *Proc. of the 5th International Workshop on Data Management on New Hardware*, DaMoN'09, pages 21–26, New York, NY, USA, 2009. ACM.

[19] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT'09, pages 24–35, New York, NY, USA, 2009. ACM.

[20] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated Control for Elastic Storage. In *Proc. of the 7th International Conference on Autonomic Computing*, ICAC'10, pages 1–10, New York, NY, USA, 2010. ACM.

[21] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware Based Data Replication Providing Snapshot Isolation. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD'05, pages 419–430, New York, NY, USA, 2005. ACM.

[22] Richard Mcougall, Wei Huang, and Ben Corrie. Cooperative memory resource mamagent via application-level balloon. Patent Application, 12 2011.

[23] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE'05, pages 13–23, New York, NY, USA, 2005. ACM.

[24] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. Technical report, 2013.

[25] Oracle Database Advanced Replication. `http://docs.oracle.com/cd/B28359_01/server.111/b28326/repmaster.htm`.

[26] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-aware storage layout for database systems. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD'10, pages 939–950, New York, NY, USA, 2010. ACM.

[27] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proc. of the 5th ACM / IFIP / USENIX International Conference on Middleware*, Middleware'04, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[28] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. Extending DBMSs with Satellite Databases. *The VLDB Journal*, 17(4):657–682, July 2008.

[29] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. OLTP on hardware islands. *Proc. VLDB Endow.*, 5(11):1447–1458, July 2012.

[30] Running SQL Server with Hyper-V Dynamic Memory. `http://msdn.microsoft.com/en-us/library/hh372970.aspx`.

[31] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application Level Ballooning for Efficient Server Consolidation. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys'13, pages 337–350, New York, NY, USA, 2013. ACM.

[32] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database Engines on Multicores, Why Parallelize when You Can Distribute? In *Proc. of the 6th Conference on Computer Systems*, EuroSys'11, pages 17–30, New York, NY, USA, 2011. ACM.

[33] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys'13, pages 351–364, New York, NY, USA, 2013. ACM.

[34] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1):7:1–7:47, February 2008.

[35] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In *Proc. of the 7th Conference on File and Storage Technologies*, FAST'09, pages 71–84, Berkeley, CA, USA, 2009. USENIX Association.

[36] TPC, `http://www.tpc.org`.

[37] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proc. of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[38] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. of 21st ACM SIGOPS symposium on Operating systems principles*, SOSP'07, pages 59–72, New York, NY, USA, 2007. ACM.

[39] Xen Hypervisor 4.1, `http://xen.org/`.

[40] Xkoto. `http://www.terradata.com/`.