ShuttleDB: Database-Aware Elasticity in the Cloud

Sean Barker [1*], *Yun Chi* [2*], *Hakan Hacıgümüş* [3], *Prashant Shenoy* [1], *Emmanuel Cecchet* [1]

[1] *School of Computer Science, University of Massachusetts Amherst, Amherst, Massachusetts, USA*

{sbarker,shenoy,cecchet}@cs.umass.edu

[2] *Square Inc., San Francisco, California, USA*

layunchi@gmail.com

[3] *NEC Laboratories America, Cupertino, California, USA*

hakan@nec-labs.com

Abstract

Motivated by the growing popularity of database-as-a-service clouds, this paper presents ShuttleDB, a holistic approach enabling flexible, automated elasticity of database tenants in the cloud. We first propose a database-aware live migration and replication method designed to work with off-the-shelf databases without any database engine modifications. We then combine these database-aware techniques with VM-level mechanisms to implement a flexible elasticity approach that can achieve efficient scale up, scale out, or scale back for diverse tenants with fluctuating workloads. Our experimental evaluation of the ShuttleDB prototype shows that by applying migration and replication techniques at the tenant level, automated elasticity can be achieved both intra- and inter-datacenter in a database agnostic way. We further show that ShuttleDB can reduce the time and data transfer needed for elasticity by 80% or more compared to tenant-oblivious approaches.

1 Introduction

In recent years, online applications have increasingly migrated to cloud platforms, which use data centers to provide computing and storage resources to these applications. Databases are no exception to this trend, and many services have been built on the idea of the "database cloud." Examples include Amazon RDS [1] and Google Cloud SQL [19], which expose cloud-based relational databases to client applications, and Salesforce, which employs a shared database used by many independent customers. There are numerous advantages to the cloud-based model, such as the pay-as-you-go model, where resources are billed and paid for on a fine-grain usage basis, and flexible resource allocation, where computing and storage can be dynamically increased or decreased based on an application's changing workload needs.

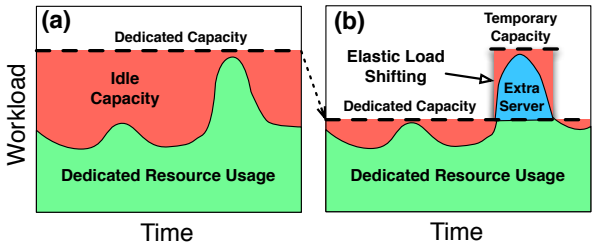

Figure 1: Provisioning for peak demand (left) may result in lower utilization and higher costs versus an elastic approach (right) requiring fewer dedicated resources.

The key to realizing many of these benefits is *elasticity* – the ability of the cloud platform to adjust an application's resource allotment on the fly when responding to long-term workload growth, seasonal variations, or sudden load spikes. Since workload peaks tend to be transient in nature, a priori static provisioning for the peak results in substantial waste as resources sit unused at non-peak times—typical server utilization in real data centers has been estimated at only 5-20% [4]. An example of this issue is shown in Figure 1. Figure 1(a) shows an example workload serviced by traditional dedicated resources. However, if the system is elastic, as shown in Figure 1(b), we can simply shift load to a new server during peak demand, and thus reduce the dedicated resources needed to service the workload.

Database Elasticity. Despite their widespread use, databases present some of the greatest difficulties in supporting elasticity. These difficulties largely stem from the specific requirements of relational databases, such as transactions and ACID compliance, and have led some to label the SQL database as the "Achilles heel of cloud elasticity" [20]. While most modern databases support clustering and replication (e.g., products such as MySQL Cluster), these systems are seldom designed to dynamically grow or shrink, and often introduce configuration,

*This work was performed while the author was at NEC Labs America. This research was supported in part by NSF grant CNS-1117221 and a gift from NEC Labs America.

management, or performance overhead [15].

While it is possible to encapsulate databases into virtual machine containers and use "standard" VM-based cloud elasticity mechanisms such as VM migration or VM replication (e.g., Amazon's auto-scaling [2]), the approach does not work well for many database cloud scenarios. For example, shared hosting scenarios (e.g., Salesforce cloud) are designed to collocate large numbers of (small) database tenants on single servers, which is not well suited to a VM-per-tenant elasticity model. If multiple tenants share single VMs to counteract the impact of hosting many VMs, then elasticity of the server is limited when using VM-level techniques, since individual tenants can no longer be migrated or replicated. VM-level mechanisms may also be unnecessarily heavyweight [24] for a "standard" application such as a database server that may not require significant customization of the underlying system.

Alternatives to VM-based black-box elasticity for databases include NoSQL systems (e.g., key-value stores such as BigTable [10] or Dynamo [16]), or augmenting the database engine itself to support elasticity (e.g., Zephyr [18], Albatross [14], and RemusDB [23]). However, these approaches introduce significant added complexity within the database itself and may change the behavior observed by client applications (e.g., the possibility of transaction failures during migrations [18]). In some ways, however, multitenant databases are well suited to application-level elasticity; for example, each database tenant may be viewed as a lightweight container, in the same way that a single VM is the unit of elasticity in VM migration.

Ideally, database elasticity should be based on migration and replication—like VM elasticity—and be transparent to the application and database engine, just as VM elasticity is transparent to the application and the OS.

Contributions. In this paper, we present ShuttleDB, a flexible system combining virtual machine elasticity with lower-level, database-aware elasticity to provide efficient database elasticity both within and across cloud data centers. Our primary contributions are threefold:

1. We examine the dichotomy between high-level VM migration and low-level DB-aware migration to decide *when* each approach is more appropriate. In particular, we identify two primary system dimensions determining what type of elasticity is appropriate—tenant type/size (i.e., collocated vs dedicated) and network type (i.e., LAN vs WAN). On this basis, for each database tenant that requires to be elastically scaled, ShuttleDB determines (a) whether to use high-level VM elasticity or low-level DB-aware elasticity, and (b) whether to scale up, scale out or scale back.

2. We present a specific technique for database-aware live migration, allowing migration of individual database tenants among servers without incurring the full cost of virtual machine migration. In contrast to most existing work, our technique requires no changes to the database engine, relying only on the presence of standard hot backup tools, and can be done live without any database down time. Additionally, performance optimizations enable efficient operation across wide-area networks.

3. We present a prototype implementation of ShuttleDB as a elasticity middleware that uses an off-the-shelf DBMS and virtualization platform. We empirically evaluate our system and demonstrate that it automatically achieves efficient elasticity under a variety of system conditions. In particular, we find that it provides tenant migration with minimal client workload delays and can reduce the time and data transfer needed for elasticity by 80% or more compared to tenant-oblivious approaches.

2 Database Clouds and Elasticity

Database Clouds. We assume the environment of a cloud providing a "database as a service" to its customers. In a database cloud, each customer rents a database from the cloud and manages it as a normal relational database, while the cloud is responsible for ensuring high performance. In particular, we assume that the cloud has a pool of physical, virtualized servers spread across one or more data centers. Each server houses one or more virtual machines (VMs), and each VM in turn houses one or more database tenants. In our work, a *tenant* is defined simply as an independent consumer of resources — most likely a single user or customer servicing a particular application. Multiple tenants may reside on individual servers so as to maximize server utilization and minimize hardware costs.

Small tenants (those with little data and/or small workloads) may be co-located within a single VM to avoid incurring the memory overhead of housing many VMs with individual OSes. Within a VM, all collocated tenants are assumed to share the same database process, which has been advocated in order to maximize resource sharing [13]. As a illustrative example, we executed a simple workload across 10 tenants on a server, first sharing an instance between all tenants (single-process), then with 10 separate processes with equal resource shares (multi-process). Transaction latencies as we scale the aggregate workload in both cases are shown in Figure 2. As the server becomes heavily loaded, we see that the multi-process model quickly falls behind, eventually showing 3x higher latency than the single-process model.

Large tenants, on the other hand, are housed in a dedicated virtual machine, which enables them to use all of the CPU and storage resources allocated to the VM. This model is analogous to EC2 compute clouds or shared hosting scenarios such as Salesforce [26].

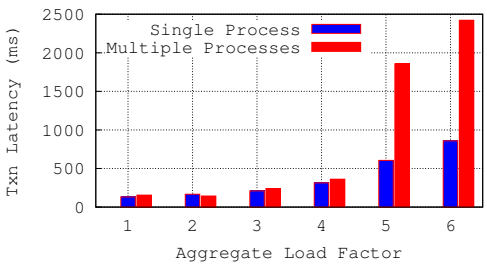

Figure 2: Single-process multitenancy outperforms multi-process multitenancy under load.

Tenant workloads are dynamic and can exhibit temporal variations or sudden spikes. In response to such variations, an important function of the cloud is to *elastically* scale the resources available to tenants. The mechanisms for providing this elasticity can be divided into two primary categories: VM-level or database-level approaches.

VM-level mechanisms: In the simplest case, when a tenant is about to experience an overload, its virtual machine can be migrated to a larger physical server and the VM can be allocated additional resources as needed. Such VM migration can be performed on-demand, transparently, and without noticeable downtime [11]. Although widely supported by virtualization platforms, however, VM migration is is wasteful for small tenants. Since many small tenants share a single virtual machine, moving the underlying VM and its disk moves *all* resident tenants, even if only one tenant is experiencing the increased workload. The simplicity of VM migration is, however, useful for large tenants housed in dedicated VMs – in this case, although the migration moves both the OS and the database, the extra overhead of moving the OS state gets increasingly amortized as the database size grows. VM mechanisms may also be used to replicate tenants onto different servers. In this case, a snapshot of the virtual machine disk is taken and copied over to the new server, which is started as a new VM instance containing a replica of the tenant Scale out via replication is best suited for large database tenants where the capacity required to service the tenant workload exceeds that of a single physical server.

Despite the simplicity and application-agnostic nature of VM-level elasticity, the VM-based approach is not without drawbacks. First, since VM migration and replication are black-box techniques, we cannot exploit useful application-specific properties, such as the presence of a database query log (which allows for *query shipping* instead of only *data shipping* as done in VM migration). Second, since most VM migration designs assume a shared, network-attached storage system, the target scenario is shipping *memory* state rather than disk state as in a database. Recent versions of Xen support migration of both VM memory and disk state in shared-nothing servers, but this still assumes a LAN environment. WAN-oriented extensions to VM migration have been proposed [29] but are not yet natively supported by off-the-shelf VM platforms.

	Dedicated Tenant (large)	**Colo Tenant** (small)
LAN scale-up	VM migrate	DB migrate
LAN scale-out	DB or VM replicate	
WAN scale-up	DB migrate	DB migrate
WAN scale-out	DB replicate	

Table 1: Elasticity mechanisms in ShuttleDB are intelligently chosen on tenant size and network characteristics.

Database-level mechanisms: A different approach is to implement elasticity at the application (i.e., database) level. Many such systems have been previously proposed [14, 18, 23], but operate largely by modifying the internals of the database itself. Such modifications complicate usage by end-users, as the operational guarantees of the database may change—e.g., the possibility of unpredictable transaction failures during migration in [14]. To be practical, database enhancement should retain transactional (ACID) semantics when implementing migration. Similarly, traditional master-slave replication allows us to implement a basic form of scale-out elasticity by adding or removing slave instances. However, many replication environments are oriented towards heavily manual configuration, and simply moving a database to a new server without downtime often involves more overhead than necessary if using replication.

Database-level elasticity has several advantages when used in cloud environments. Since replication and migration can be performed on a per-tenant basis, it is particularly useful for small tenants that share a virtual machine (as overloaded tenants can be individually migrated). For large tenants, these differences are less important and the benefits of database-level elasticity may not outweigh the downsides of added complexity. For cross-data center WAN elasticity, however, database-level mechanisms are still preferable due to the current limitations of VM-level mechanisms over WAN migration.

ShuttleDB approach. From the previous discussion, it follows that neither VM-level nor database-level elasticity works well in all scenarios. Consequently, ShuttleDB incorporates both VM-level and DB-level elasticity and automatically chooses the "best" elasticity mechanism for each elastic operation on a given tenant (e.g, VM-level or DB-level, and scale-out or scale-up). In addition, ShuttleDB provides its own technique for database live migration that does not require any modification to the internals of a database and works with most off-the-shelf database platforms. We also show how this approach can be used to implement database replication in multi-tenant master-slave settings.

The ShuttleDB approach is summarized in Table 1. For small, co-located tenants within a VM, scale up is the best elasticity option, since tenant requirements are less than the capacity of a single server. Further, database-level mechanisms are more efficient since they enable a single tenant to be migrated independently of others, while a VM-level mechanism would need to move the entire VM and all resident tenants. For large dedicated tenants, the choice depends on other factors. If the tenant is still smaller than a single machine, scaling up by migration to a larger machine is a feasible option. Within a LAN, VM migration is the simplest approach. When the tenant requirements exceed the capacity of a single server, it must be scaled out by replicating onto multiple machines. Within a LAN, this can be achieved by either VM- or DB-replication. Across cloud data centers, however, DB-level migration or replication is preferable to VM-level mechanisms, due to the premium placed on bandwidth usage in such scenarios.

3 Database-Aware Elasticity

Elasticity requires both migration for scaling up and replication for scaling out. Replication is actually a specific case of migration where the original database does not have to be stopped at the end of migration. Virtual machine migration is built-in to modern hypervisors, and as such is relatively simple to employ. Database-aware migration, on the other hand, is not part of off-the-shelf DBMSes. To address this issue, ShuttleDB provides its own database migration technique, which also serves as the building block for replication elasticity. Our technique draws inspiration from live VM migration [11] and database migration [6] methods, but has important differences from both. While VM migration uses "black box" data shipping to transfer state to the target machine, our DB migration technique combines query log shipping and data shipping for greater efficiency. "Process-level" techniques used in previous work such as [6] migrate *all* database state managed by a database server process, while our method can perform "tenant-level" migration where individual tenant databases within a database server can be live migrated. ShuttleDB's elasticity protocol is 'live', with minimal downtime regardless of data size, and is fully transparent to clients of the database. Our database-agnostic migration protocol is described below, while our database-specific prototype implementation is detailed in Section 5.

3.1 Migration Protocol

DB-aware migration in ShuttleDB employs a three-phase database migration protocol, which is shown in Figure 3. For a given migration, we have three logical machines to consider: the source server on which the migrating tenant currently resides, the target server to which the tenant is migrating, and the client(s) currently issuing requests to the migrating tenant.

Phase 1 – Hot Backup. In the first phase, we create a *live snapshot* of the tenant database by employing an off-the-shelf hot backup tool and streaming the resulting backup image to the target server. Suitable hot backup tools are available for most well-known database systems (MySQL, Postgres, Oracle, etc.). Since the tenant may have a large amount of data, taking this snapshot may take minutes to hours. However, as the server is not blocked during this period, it continues to service all client workloads as usual. At the completion of the first phase, the target server contains a copy of the tenant database up to some position in the binary log.

Phase 2 – Live Deltas. Once phase 1 completes, a consistent snapshot of the tenant exists on the destination server, but may be out of date, since the local server continues to execute queries during the backup. If replication has to be performed, the slave replica can be started right away at the destination, letting the database replication mechanisms bring the replica up to date.

In case of migration, the second phase proceeds as a series of *delta rounds* in which the source server replays queries from the binary log and streams them across the network to bring the target server up to date. This is analogous to memory deltas used in VM migration, but applied to a query log instead of the contents of memory.

Let p_s be the log position of the source server and p_t be the log position of the target server. At the start of each round, the target server sends p_t to the source. The source then reads from p_t to p_s in the log and sends this delta to the target, which is applied to the target database after filtering queries not pertaining to the migrating tenant. Once the delta is applied to the target, the next round begins by again sending p_t to the source.

Delta rounds continue until the duration of the most recent round is either (a) less than a small threshold (e.g., a few seconds) or (b) greater than the round before it. This approach guarantees termination of phase 2. In typical circumstances, each delta round is shorter than the last (since the number of write queries in the next delta round is proportional to the duration of the previous round), which will ultimately result in satisfying condition (a). If this is not the case – i.e., delta rounds are getting longer – then this is an indication that the target server is actually falling further behind the source and will trigger condition (b). In either case, migration proceeds to phase 3.

Phase 3 – Handover Delta. Once delta rounds are completed, the two copies of the database are nearly in sync, and we are ready to hand off the tenant workload from the source to the target. To do this, a final *handover delta* is performed to complete migration. The lo-

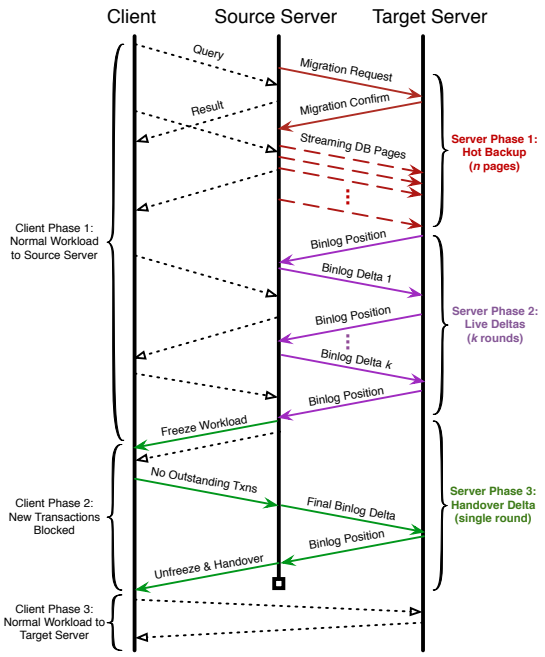

Figure 3: Three-phase live migration protocol for database tenants used in ShuttleDB.

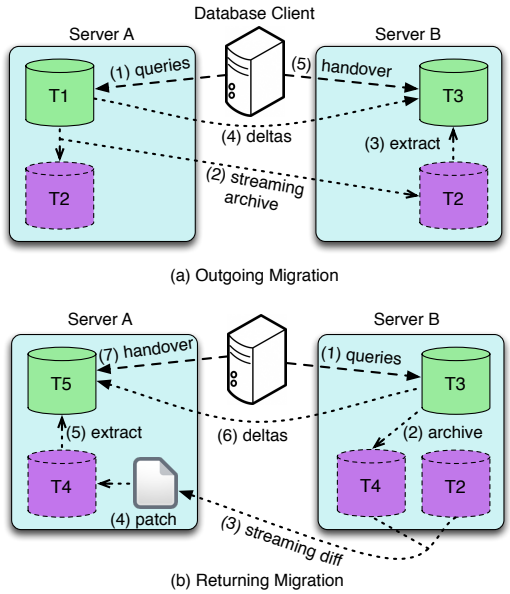

Figure 4: Saving a tenant archive during migration allows transferring small diffs instead of the full tenant database during subsequent migrations.

cal server first freezes the tenant workload by queueing all new incoming transactions directed at the migration tenant (but without blocking existing transactions), then waits to finish servicing the tenant's outstanding transactions prior to the freeze. Once all such transactions are completed, the local server sends a final delta, bringing the target server fully in sync with the source (since new transactions are frozen in the meantime). Once the final delta is applied, previously queued transactions are forwarded to the target server, clients are redirected to the new server, and the tenant workload is unfrozen. This completes the handover and switches the 'authoritative' tenant copy to the target server. Note that while phase 3 necessarily imposes a degree of effective downtime while the final delta is applied, this duration is *not* dependent on the size of the database and is typically a second or two at most, as demonstrated in Section 6. Downtime may be longer if phase 2 was terminated by lengthening delta rounds, but this case is unlikely except with extremely write-intensive workloads.

To adjust this migration protocol for replication, we do not need to freeze the workload at the master. Instead, the slave connects to the master and receives a continuous stream of updates from the master. Queries can be directed to the new slave as soon as it has caught up.

3.2 Performance Optimizations

We employ two notable performance optimizations during the migration process. First, we consider the fact that since migration itself imposes some degree of overhead (e.g., streaming the migrating tenant), local performance may be degraded. To address this issue, we employee an automated technique previously proposed [6] that automatically and dynamically rate-limits migration (depending on performance) to avoid excessive interference. Second, we consider the case when a migration is performed to a server containing an old version of the tenant (e.g., when migrating to a server containing a backup or returning from a cloud server following a temporary workload spike). Here, we do not need to migrate the *entire* tenant, but can transmit only the (small) delta between the old tenant copy and the up-to-date version.

This process, which we term 'delta migration', is illustrated in Figures 4(a) and (b) for an outgoing and returning migration, respectively. During the initial hot snapshot (phase 1) of the outgoing migration, we save a copy of the snapshot on both the source server (a local copy) in addition to the usual streaming copy to the target server. The remainder of phases 2 and 3 then proceed as normal. During phase 1 of the return migration, however, rather than streaming the database back to the source server, we re-use the older snapshot already present locally. We then generate a patch from the original database snapshot and stream only this patch back to the source. Since the source still has a copy of the original snapshot, it then applies the patch to generate the up-to-date snapshot, then proceeds phases 2 and 3 of bursting as usual.

In performing this optimization, we are able to skip the most expensive part of migration (the full data stream in phase 1). The net result is a major reduction in the

amount of network data that must be transferred to migrate between machines. This reduction in network traffic is of particular interest when migrating over the wide-area, e.g., between a local cluster and Amazon EC2, since such transfers are likely to be over relatively low-bandwidth links.

Note that pre-copying can also be employed to reactively spawn slave replicas in case of a sudden load spike. Previously terminated replicas can be quickly restarted by sending the small diff patch containing the updates that occurred since they were stopped.

4 Automated Elasticity

ShuttleDB employs an intelligent algorithm that combines VM-level elasticity with the database-level techniques described in the previous section to automatically scale the capacity of each tenant as needed by the workload. The algorithm involves four key steps: (i) *when* to invoke the scaling algorithm, (ii) *who* (i.e., which tenant(s)) to choose for optimization, (iii) *where* to migrate or replicate the tenant(s), and (iv) *which* mechanisms to use for scaling: i.e., scale-up, scale-out or scale-back and whether to use VM-level or DB-level techniques. The algorithm involves the following steps:

Step 1: **When** *to initiate elastic scaling:* ShuttleDB monitors the current query latency of tenants in the database server (computed as a smoothed average over a sliding window) and also tracks the resource usages at the underlying virtual machine to determine when to initiate the scaling algorithm. ShuttleDB uses an upper threshold on latency and resource utilization as well as a lower threshold on these values to initiate scale-up/out and scale-back, respectively. Further, in addition to *reactively* triggering the algorithm when the thresholds are breached, it is also possible to use time series based load forecasting that uses past trend history to predict future values and use these predictions to *proactively* initiate the algorithm. Currently, we use a standard ARIMA time-series forecasting method, which smooths the observed latencies and utilization over recent time periods to predict future values [28].

Step 2: **Which** *tenants to choose for scaling:* In most cases, the tenant that is experiencing the overload (or is about to, as per predictions) is chosen for scaling (or for scaling back if it is under-loaded and below the low threshold). However the choice of which tenants to choose may not always be staightforwad. In certain shared co-location scenarios, for example, no single tenant may be experiencing an overload but each tenants may be experiencing small increases in load so that they all collectively exceed the higher threshold. A more interesting scenario is one where one tenant is overloaded, but it may be *cheaper to move out a different tenant* and give the freed resources to the overloaded tenant. For example, if two tenants equally share a VM's CPU and memory and have database sizes of 5GB and 10GB, and if latter experiences overload, it is cheaper to move the first tenant and give all of the VM's resources to the latter. To intelligently choose the "correct" tenants, ShuttleDB performs a simple cost-benefit analysis. The *cost* of moving a tenant is estimated as the amount of disk state (and possibly memory state when using VM-level migrations) that must be transferred. The *benefit* of moving a tenant is the amount of load it offloads to a different server (measured as CPU or disk load, depending on the bottleneck resource on that VM). ShuttleDB greedily chooses the tenants with the greatest benefit to cost ratio – that is, those that offload the most load at the lowest data transfer cost.

Step 3: **Where** *to move a tenant:* Whenever possible ShuttleDB attempts to move tenants to servers in the same cloud data center. In scenarios where local resources are stressed, ShuttleDB will then choose to move tenants to servers in the nearest cloud site. Note that such WAN-level migrations or replication must be done carefully since there will be an impact on the front-end tiers of the application, which may experience WAN latencies if the backend tenant moves to a different site. Typically such moves would be done in consultation with the elasticity mechanisms of the front-tier as well, but such coordinated elasticity mechanisms are beyond the scope of this paper, and here we assume that the scaling of database cloud is done independently.[1] Typically for scale out and scale up, ShuttleDB chooses any server with sufficient idle resources. Scale back for shared tenants involves a consolidation and ShuttleDB chooses a virtual machine that hosts other shared tenants but has sufficient resources to house more.

Step 4: **Which** *mechanisms to choose:* The final step involves determining which elasticity mechanism to choose. Table 1 depicts the preferred DB or VM-level mechanism used in each case. For shared (small) tenants, scaling up is the preferred option and DB-level mechanisms are used to move only the desired tenants and avoid needless data copying. The only scenario where scale out is used for a small tenant is when it experiences a very large workload growth - in this case, a DB migration is first performed to extract the tenant out of the shared VM and it is given its own VM, which is then replicated, like in the large dedicated tenant case. For large dedicated tenants, ShuttleDB attempts to scale up (migrate to bigger server) when possible and then uses scale out (replicate to other servers) when no single server can service the incoming workload. LAN mechanisms are always preferred over WAN.

[1] Amazon's S3 storage also performs such geographic replication independently, although latency issues are more critical for databases.

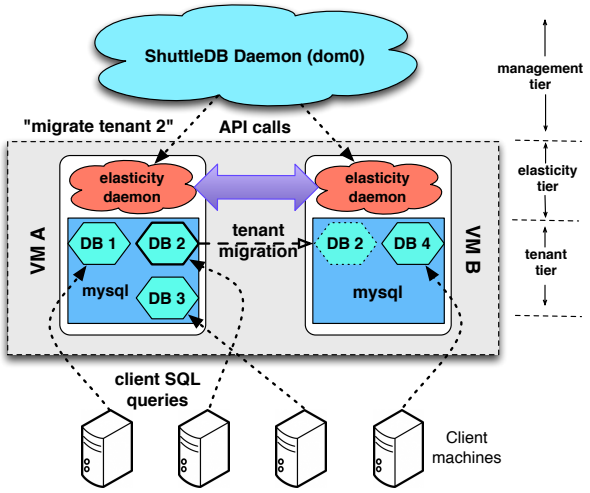

Figure 5: ShuttleDB consists of three tiers: a database/tenant tier (using an off-the-shelf DBMS), a database elasticity tier (using VM or DB-aware migration), and a top-level manager (e.g., to facilitate elasticity).

5 Prototype Implementation

Our current implementation of ShuttleDB uses a three-tier design as depicted in Figure 5. End-users (or application servers) talking to the databases interact with the lowest level of the system, which are standard database processes running within VMs (e.g., `mysqld`). Above the database servers is the elasticity layer, which provides the automated mechanisms to dynamically grow, shrink, or relocate the resources allocated to any given database tenant, using either VM-level or DB-level elasticity. At the highest level is the elastic database manager, which employs the simple API exposed by the elasticity layer to manage server and tenant workloads. The manager may implement simple or sophisticated algorithms for automatically administrating a database server cluster. While many database managers are possible, we present an example in Section 6 for automating the process of 'cloud bursting', in which tenants are migrated to remote servers to alleviate workload spikes.

Each physical server in our prototype runs a single instance of the ShuttleDB daemon on the domain0 VM, which runs Xen Cloud Platform 1.6 on top of CentOS 6 to manage all VMs on the server. Multiple daemons communicate in a peer-to-peer fashion to facilitate migrations. Separate daemons runs within each domainU VM to manage the database-aware elasticity layer.

VM elasticity. To allow for shared-nothing live migration of virtual machines, we make use of the Storage XenMotion feature, which allows for moving disk, memory, and virtual devices to a remote host. To clone an active VM, we snapshot the VM (a live operation), then export the snapshot to a new VM on the local or remote host. We then start the cloned VM and update its network settings to result in a suitable VM for replication.

DB elasticity. We implemented our technique for database-aware migration on top of MySQL, using the Percona XtraDB [25] database engine (effectively InnoDB with a few useful extensions pertaining to taking backups) provided by the Percona Server package. The database elasticity implementation is loosely coupled from the database engine itself, as follows. For migration phase 1 (streaming backups), we use a hot backup tool (Percona `xtrabackup` [30] in the current prototype) to snapshot and extract the data for the migrating tenant, which is compressed and streamed across the network to the target server. On completion, the target server imports the tenant database into the already running database server (this functionality is provided both by XtraDB and bleeding-edge versions of InnoDB [5]). We perform phase 2 (live deltas) by reading from the database transaction log and streaming updates to the target server. The queries executed since the initial snapshot or last delta round are filtered to remove extraneous queries pertaining to non-migrating tenants, then shipped to the target server and executed. For simplicity, once the handover of phase 3 begins, we employ a client-side proxy to freeze the workload and temporarily queue transactions, then release those transactions to the target server once the handover completes. This approach avoids having to make any modifications to the database engine itself.

Delta migrations. Our implementation handles the delta migration optimization described in Section 3.2 at a disk block level. During the initial outgoing migration, the local copy of the tenant is saved as a single binary archive file, which is also transferred to the target server. When the return migration is initiated, the database patch is simply generated as a disk-level patch between the original archive file and the up-to-date archive file using `rdiff`. Since this approach is oblivious to any actual data formats used by the database, it relies on block-level similarities between the archives to generate a compact diff – however, our experiments in Section 6 demonstrate that this is generally sufficient and results in a small patch file. One issue we encountered during testing was workload interference resulting from the archiving process, since saving a local copy of the archive required substantial I/O resources. To counteract this, we added the option to dedicate either a spare disk or a RAM-based volume for archiving operations. Local archiving may also be disabled entirely, which removes much of the I/O overhead of migration but prevents use of the delta migration optimization – effectively trading off between disk and network resource consumption.

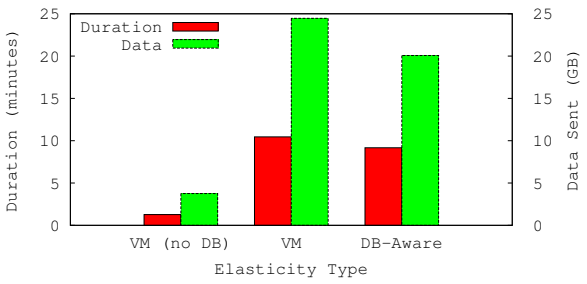

Figure 6: LAN migration of a large (dedicated) tenant.

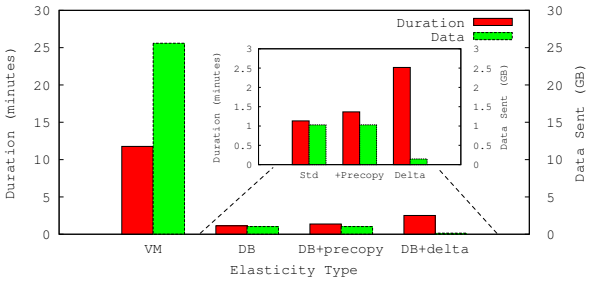

Figure 7: LAN migration of a small (collocated) tenant.

6 Experimental Evaluation

To evaluate our prototype, we first consider the individual elasticity mechanisms employed by ShuttleDB (from Table 1), then consider two scenarios demonstrating ShuttleDB's utility – first, in automating database cloud bursting, and second, in leveraging DB-aware elasticity to increase the efficiency of VM replication.

We use a heavily modified version of the Yahoo Cloud Serving Benchmark (YCSB) [12] to generate the workload for our system. While YCSB was originally designed exclusively for key-value stores, we begin with an extended, transactional version used in prior work [17, 14, 6] and further extend it to generate a closed workload with Poisson-distributed arrival times. Each workload consists of replaying a trace of workload 'intensities', which determine the number of transactions issued to the tenant per time unit.

6.1 LAN Elasticity

We first compare the efficiency of VM and DB-aware migration when operating over a LAN, in order to substantiate our earlier arguments about when each technique is preferable. We configure a VM with 30 GB of storage and 2 GB of RAM. The size of the base system is roughly 1.6 GB, while all additional space is used by the database server. We first consider moving a large (i.e., dedicated) database tenant by configuring a 20 GB tenant and moving it to a second server while servicing a workload. Figure 6 shows the duration of migration for 3 cases: a no-tenant baseline (i.e., only the OS), VM-based tenant migration, and DB-aware tenant migration. We see that the benefit of employing DB-aware elasticity in this case is minimal and likely does not justify the added migration complexity versus simply using VM elasticity.

Next, we configure the VM with twenty 1 GB tenants instead of a single 20 GB tenant, and evaluate four scenarios: VM migration, DB-aware migration, DB-aware migration with precopying (i.e., preparation for future delta migrations), and DB-aware delta migration (i.e., migrating to a server with a local precopy). In each scenario, we transfer a **single** tenant from the shared server to a separate, dedicated server. In this case, as shown in Figure 7, the differences are striking. VM migration, which simply transfers the entire VM, results in nearly 20x slower migration than DB-aware migration. Adding precopying to DB-aware migration adds a small amount of overhead, but still greatly outperforms VM elasticity. Finally, using a previous precopy to perform a delta migration takes somewhat longer on account of processing the database delta, but results in transferring less than 200 MB total of data – roughly one fifth of the already reduced amount in the base DB-aware case.

Result: *With large tenants, the simplicity of VM migration is preferable. With small tenants, however, DB-aware elasticity greatly outperforms VM elasticity. The use of precopying can even further reduce the amount of network data required.*

6.2 DB-Aware Live Migration

Next, we evaluate the 'liveness' of our DB-aware migration technique by considering the amount of downtime incurred. We configured two ShuttleDB servers over a LAN, and a single tenant with 1.5 GB of data servicing 80 read and 20 write queries per second. We then moved the tenant from the source server to the target using DB-aware elasticity, observing the transaction latencies shown in Figure 8. Migration begins at event (a), at which point ShuttleDB begins streaming the tenant to the target server (phase 1). As seen, this operation has little to no visible impact on tenant performance. Less than 3 minutes later, at event (b), the initial streaming copy is completed, and the target server begins preparing the copy to service the workload. Once complete, two copies of the tenant are running, and the original copy begins applying deltas to the new copy (phase 2). Delta rounds take roughly 15 seconds, at which point the workload is frozen and the handover (phase 3) is performed.

The inset of Figure 8 shows a plot of transaction latencies over time around the time of the handover. Latency just following the handover spikes, owing to the frozen workload (during which transactions are queued

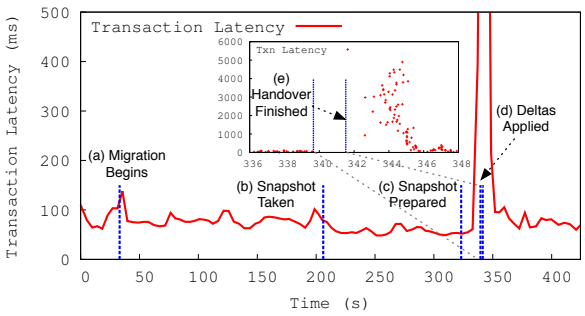

Figure 8: Live migration of a tenant servicing 10 transactions (100 queries) per second.

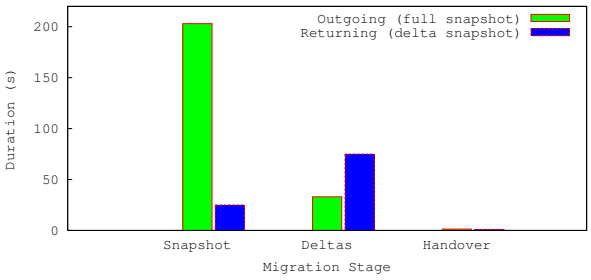

Figure 9: DB-aware delta migrations significantly reduce bandwidth and time requirements across wide-area networks.

at the source). However, this period lasts only 2 seconds, after which queued transactions are released to the target. Here, the queued transactions are delayed by an average of 2-3 seconds, but only 25 transactions in total are affected, and the entire duration of possible delays lasts less than 5 seconds. Finally, we note that this result is conservative, since the migrating tenant is servicing many queries during the handover; less active tenants will experience a shorter handover period, and thus will observe lesser delays. As seen, the only notable workload impact occurs during this handover period, whose duration is dependent *only* on the write workload intensity, and is not related to data size. Migrating a larger tenant simply extends the harmless duration of phase 1.

Result: *DB-aware elasticity in ShuttleDB provides robust migration capabilities with near-zero downtime and minimal query delays, even for highly active tenants.*

6.3 Wide-Area Elasticity

Elasticity between data centers (i.e., over a WAN) is challenging due to lower bandwidth and higher latencies. To evaluate this scenario, we configured a source server on the west coast of the US with ten 512 MB database tenants handling 50 queries per second, and a target server in an Amazon EC2 data center located on the east coast of the US. As shown in Figure 9, we then shifted one of the ten tenants to the EC2 server using DB-aware elasticity, then back from the EC2 server using a delta migration after waiting several minutes. As shown, nearly 90% of the elapsed time of the initial migration is spent transferring the initial snapshot (unsurprising given the limited available bandwidth). Importantly, the entire period in which the workload is **not** serviced (during the handover phase) lasts only a single second.

On the returning migration, we again observe the effectiveness of DB-aware delta migrations. While processing deltas increases the time spent in phase 2, the decrease in phase 1 more than compensates, and the total migration duration on the return is less than half that of the outgoing migration. Furthermore, the amount of data transferred during the delta migration is only 97 MB, as compared to 776 MB during the outgoing migration – over an 87% reduction.

We could not easily perform a companion experiment using VM elasticity, as public data centers such as EC2 do not expose their hypervisor infrastructure. Despite this, however, we would not expect VM elasticity to perform well given the limitations demonstrated in the previous experiment. Moreover, over a lower-bandwidth network such as a WAN, we would expect the differences to be even more significant than before.

Result: *ShuttleDB elasticity can effectively span multiple networks and data centers and minimizes the data transfers necessary.*

6.4 Defusing a Hotspot

Here, we present end-to-end experiments demonstrating how ShuttleDB responds to a server hotspot by migrating and replicating tenants. We use the World Cup soccer trace [3] to generate the workload for our experiments. This trace contains an end-to-end workload hotspot, starting with a stable (low) arrival rate, rising to a peak, then falling back to the baseline.

We configured our local server with 10 tenants, each with 1 GB of data. The query arrival rate of a tenant is driven by the world cup trace, while the other tenants run a standard arrival rate – initially, all arrival rates are identical. We set the bursting threshold latencies to 300 ms (upper) and 200 ms (lower). The multi-query transactions executed by all tenants include a mix of read, write, sort, and join operations, with specific arrival times given according to a Poisson distribution. Finally, the duration of the world cup trace is scaled to 90 minutes.

First, we run the workloads with ShuttleDB disabled, to observe the effects of the hotspot on tenant performance. We then re-run the same experiment with tenant migration on LAN and WAN. All the results are summarized in Figure 10. The increase in workload around $t = 50$, results in a sharp latency increase that exceeds

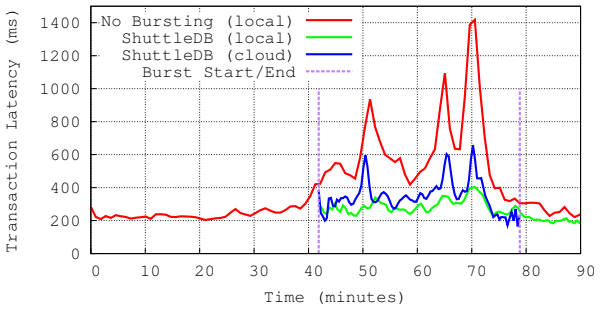

Figure 10: ShuttleDB automatically migrates tenants to mitigate the impact of a workload spike.

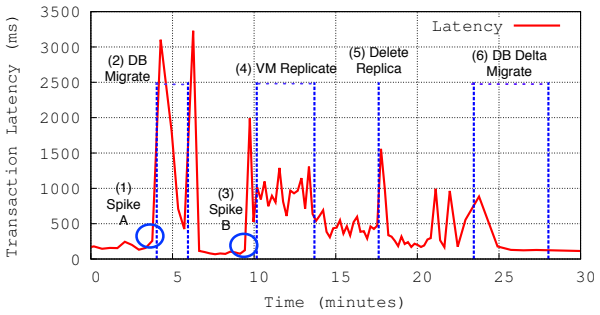

Figure 11: Database-aware elasticity with two workload spikes. First spike handled with tenant migration, and second spike with replication before scaling back.

4x baseline performance without ShuttleDB. When running with ShuttleDB, however, a migration is initiated around $t = 70$, once the ARIMA prediction determines that latency will exceed the threshold value of 300 ms. Whether the tenant is migrated to another database on the same LAN or over WAN, the latency remains within reasonable bounds with an average latency increase of 100ms for the LAN case. Once the peak has passed and latency begins to fall to normal levels (around $t = 150$), the tenant is returned to the local server.

In a second experiment, shown in Figure 11, we put the tenant under a series of two workload spikes, exceeding the capacity of the multitenant server. Following the spikes, the workload on the tenant gradually reduces to normal. To address the workload spikes, we provisioned two spare servers for ShuttleDB to use.

After the first spike, ShuttleDB migrates the single tenant out of the multitenant server to the first dedicated server, stabilizing performance. Note that although latency spikes briefly immediately following this migration due to a cold cache, this issue may be mitigated by executing read queries on both machines to warm the target cache prior to the handover. After the second spike, latency increases yet again, and so ShuttleDB replicates the entire VM to the second spare server. However, note that the primary reason we can effectively employ VM replication is *because* of the migration, which extracts the single tenant. Following the spikes and gradual decrease of the workload, the database is able to scale back, first by deleting the VM replica, then by performing a DB-aware delta migration back to the original server.

Result: *Migration and replication can be combined in ShuttleDB to maximize elasticity in multitenant servers.*

7 Related Work

Elastic cloud platforms have been proposed for many useful applications, such as video streaming services [32] and medical image registration [21]. The general concept of 'cloud bursting' [7, 22] has become popular as a way to merge existing infrastructure with newly available cloud resources. Systems such as Dolly [9] have considered cloud systems for databases through the use of cost models governing database provisioning.

Live migration has been extensively studied in the context of virtual machines [11, 8, 29], where the key challenge is migrating a dynamic memory image with minimal downtime. Live migration has been extended to the domain of databases in the context of both shared-nothing systems [17, 18] and systems with networked attached storage [14]. Our own prior work has addressed performance interference when migrating databases [6].

Multitenant databases have also attracted significant attention due to the rise of cloud computing, at varying levels of multitenancy [17, 27, 31]. Prior work has demonstrated that purely VM-based multitenancy may result in high overhead and low tenant consolidation [13], a conclusion supported by our own studies.

8 Conclusions

In this paper, we presented techniques to implement database-aware elasticity in multi-tenant database clouds. We proposed a database-aware live migration and replication approach that is designed to work with common off-the-shelf databases without requiring any database engine modifications. ShuttleDB combines database-aware techniques with VM-level mechanisms to implement a flexible approach to achieving efficient scale up, scale out or scale back for diverse scenarios ranging from different tenants sizes to inter- and intra-data center elasticity. We implemented a prototype of ShuttleDB and experimentally demonstrated the benefits of ShuttleDB's database-aware elasticity mechanisms and intelligent elasticity algorithm. As future work, we plan to study the interplay between ShuttleDB's decision making and the elasticity mechanisms employed by other application tiers.

References

[1] AMAZON. Amazon relational database service. http://aws.amazon.com/rds/, 2013.

[2] AMAZON. Ec2 auto scaling. http://aws.amazon.com/autoscaling/, 2013.

[3] ARLITT, M., AND JIN, T. A workload characterization study of the 1998 world cup web site. *Network, IEEE 14*, 3 (2000), 30–37.

[4] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R. H., KONWINSKI, A., LEE, G., PATTERSON, D. A., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Commun. ACM 53*, 4 (2010), 50–58.

[5] BAINS, S. Innodb transportable tablespaces. http://blogs.innodb.com/wp/2012/04/innodb-transportable-tablespaces/, 2012.

[6] BARKER, S., CHI, Y., MOON, H. J., HACIGÜMÜŞ, H., AND SHENOY, P. 'cut me some slack': latency-aware live migration for databases. In *EDBT* (2012).

[7] BARR, J. Cloudbursting—hybrid application hosting. http://aws.typepad.com/aws/2008/08/cloudbursting-.html, 2008.

[8] BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. Live wide-area migration of virtual machines including local persistent state. In *VEE* (2007).

[9] CECCHET, E., SINGH, R., SHARMA, U., AND SHENOY, P. J. Dolly: virtualization-driven database provisioning for the cloud. In *VEE* (2011).

[10] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *OSDI* (2006).

[11] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI* (2005).

[12] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *SoCC* (2010).

[13] CURINO, C., JONES, E. P. C., MADDEN, S., AND BALAKRISHNAN, H. Workload-aware database monitoring and consolidation. In *SIGMOD* (2011).

[14] DAS, S., NISHIMURA, S., AGRAWAL, D., AND ABBADI, A. E. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB 4*, 8 (2011), 494–505.

[15] DATASTAX. Why migrate from mysql to cassandra. White paper, Datastax, 2012.

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *SOSP* (2007).

[17] ELMORE, A. J., DAS, S., AGRAWAL, D., AND EL ABBADI, A. Who's driving this cloud? towards efficient migration for elastic and autonomic multitenant databases. Tech. Rep. CS-2010-05, UCSB, 2010.

[18] ELMORE, A. J., DAS, S., AGRAWAL, D., AND EL ABBADI, A. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD* (2011).

[19] GOOGLE. Google cloud sql. https://developers.google.com/cloud-sql/, 2013.

[20] HOGAN, M. Cloud elasticity and databases. http://scaledb.blogspot.com/2011/08/cloud-elasticity-databases.html, 2011.

[21] KIM, H., PARASHAR, M., FORAN, D. J., AND YANG, L. Investigating the use of autonomic cloudbursts for high-throughput medical image registration. In *GRID* (2009).

[22] MELL, P., AND GRANCE, T. The NIST definition of cloud computing, September 2011.

[23] MINHAS, U. F., RAJAGOPALAN, S., CULLY, B., ABOULNAGA, A., SALEM, K., AND WARFIELD, A. Remusdb: transparent high availability for database systems. *VLDB J. 22*, 1 (2013), 29–45.

[24] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.* (Dec. 2002).

[25] PERCONA. The percona xtradb storage engine. http://www.percona.com/docs/wiki/Percona-XtraDB:start, 2013.

[26] SALESFORCE. Salesforce crm and cloud computing. http://salesforce.com, 2013.

[27] SOROR, A. A., MINHAS, U. F., ABOULNAGA, A., SALEM, K., KOKOSIELIS, P., AND KAMATH, S. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst. 35*, 1 (Feb. 2010), 7:1–7:47.

[28] WEI, W. W. S. *Time series analysis - univariate and multivariate methods*. Addison-Wesley, 1989.

[29] WOOD, T., RAMAKRISHNAN, K. K., SHENOY, P., AND VAN DER MERWE, J. Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines. *SIGPLAN Not.* (2011).

[30] Percona XtraBackup. http://www.percona.com/software/percona-xtrabackup/, 2013.

[31] XIONG, P., CHI, Y., ZHU, S., MOON, H. J., PU, C., AND HACIGÜMÜŞ, H. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE* (2011).

[32] ZHANG, H., JIANG, G., YOSHIHIRA, K., CHEN, H., AND SAXENA, A. Intelligent workload factoring for a hybrid cloud computing model. In *SERVICES* (2009).