Dolly: Database Provisioning for the Cloud

Emmanuel Cecchet, Rahul Singh, Upendra Sharma, Prashant Shenoy
University of Massachusetts, Amherst
USA
{cecchet,rahul,upendra,shenoy}@cs.umass.edu

ABSTRACT
The Cloud is an increasingly popular platform for e-commerce applications that can be scaled on-demand in a very cost effective way. Dynamic provisioning is used to autonomously add capacity in multi-tier cloud-based applications that see workload increases. While many solutions exist to provision tiers with little or no state in applications, the database tier remains problematic for dynamic provisioning due to the need to replicate its large disk state.

In this paper, we analyze the challenges of provisioning shared-nothing replicated databases in the cloud. We evaluate various replica spawning techniques and argue that being able to determine state replication time is crucial for provisioning databases. We propose Dolly, a database provisioning system based on a virtual machine cloning technique to spawn database replicas in the cloud. We propose cost models to adapt the provisioning policy to the cloud infrastructure specifics and application requirements. We present an implementation of Dolly in a commercial-grade replication middleware and evaluate database provisioning strategies for a TPC-W workload on a private cloud and on Amazon EC2. By being aware of state replication cost, Dolly can do better automated provisioning for replicated databases on cloud platforms.

1. INTRODUCTION
Online applications have become popular in a variety of domains such as e-retail, banking, finance, news, and social networking. Many online cloud applications employ a multi-tier architecture with a database back-end and a web-based front-end. Such multi-tier applications run in data-centers or on cloud computing platforms, which provide storage and computing resources to the applications. Modern data centers that run private or public cloud-based hosting platforms employ a virtualized architecture, where each tier or application component runs inside virtual machines that are mapped onto physical servers of the system.

1.1 Provisioning in the cloud
Numerous studies have shown that the workloads seen by online web-based cloud applications are highly dynamic and exhibit variations at different time-scales [29], [30]. For instance, an application may see a rapid increase in its popularity, causing its workload to grow sharply over a period of days or weeks. At shorter time-scales, a flash crowd can cause the application workload to surge within minutes. Applications can also see seasonal trends such as higher workloads during particular periods, e.g., during Black Friday, marketing campaigns or a new product launch. These workload fluctuations have to be handled by provisioning enough capacity for the application at the time it needs it.

Replication is a popular approach for dynamically provisioning capacity in private and public clouds by spawning new replicas at each tier of the application. Much of the prior work on dynamic provisioning has focused on the web or the application tiers and not on the database tier [28], [29], [30], [6], [10]. Provisioning these front-end tiers only requires starting up new web or application server processes on new machines and since much of the application's persistent state is stored in the database tier, these provisioning techniques for the front-end tiers do not need to consider the more challenging tasks of replicating this persistent state. In contrast, provisioning of the back-end database tier must not only consider startup of new database server replicas but also replication and synchronization of the associated disk state of the database.

In this paper, we consider the problem of dynamically spawning new replicas at the database tier in order to scale the capacity of multi-tier applications. We assume a shared nothing architecture, where each replica has a local copy of the database (these local copies are assumed to always be consistent with one another). Such shared-nothing databases are typical in today's cloud platforms, since shared disk architectures, such as Oracle RAC [18] require specific hardware that is usually not available in typical cloud platforms. Database provisioning typically involves the task of capacity determination of how many replicas are needed for a workload and that of the actual replication i.e. starting up new replicas. The capacity determination problem for database provisioning has been studied in [11] where analytical models for determining the capacity needed to service a given workload were proposed; this work does not consider the "hard" problem of dynamically replicating database content to startup new replicas.

Replication for shared-nothing architectures is well-studied in the literature [17], [4], [22], [12], starting with the seminal work by Gray et. al [14] that articulated the scalability issues in asynchronous replication. However, much of this database replication work has focused on performance aspects and consistency tradeoffs of transaction execution, and the global impact of management operations such as adding a new replica has been overlooked [9]. In other words, prior work on database replication has focused on scalability or consistency issues in systems with a *static* number of replicas, and the problem of dynamically scaling capacity by adding new replicas on-the-fly has received relatively little attention. This problem is important in cloud environments where dynamic workload changes require these new database replicas to be started up on-the-fly at short timescales.

1.2 Why is database provisioning hard?

Database provisioning requires a capacity determination model to estimate how many replicas to provision for a given workload. Such models predict the future workload using historical data, administrator input or dynamic predictors [16] and then use queuing techniques to estimate the capacity needed to service the predicted workload [13]. Traditional front-end provisioning techniques assume that this capacity can be added immediately (by starting up new replicas on new machines) to address workload variations. In the case of back-end database tier however, adding a new database replica involves (i) extracting database content from an existing replica, (ii) restoring that content on a new replica and (iii) synchronizing the state of the new replica with the current state of all other replicas to preserve data integrity. These operations can take minutes or hours depending on the database size.

Thus, database provisioning is very different from traditional web server provisioning because databases are stateful and their state can be very large (and this state must be replicated before a new database replica can be spawned). To provision database replicas in a timely fashion, it is necessary to know how much time will be required to replicate/synchronize this disk state and bring the replicas online. These times vary greatly with the database size, schema complexity, backup/restore tool options, database artifacts (i.e. storage engine configuration, triggers…). Moreover, there are many tradeoffs on how and when to snapshot the database state to minimize replica resynchronization time. It is therefore non-trivial to estimate the exact time needed to spawn a new replica, since it depends on many parameters.

Traditional "just-in-time" web provisioning techniques such as Amazon Auto Scaling [1] are based on thresholds and do not take into account the time to replicate the disk state. If this disk state replication overhead is ignored, the newly provisioned capacity comes online far too late to handle the workload increase and the capacity requirements will not be met in a timely fashion. As we will show in this paper, Dolly is able to provision an adequate capacity because it can estimate the time to bring a replica online and take this overhead into account by triggering the appropriate operations on time.

1.3 Contributions

In this paper, we present Dolly[1], a replicated database provisioning system for the cloud based on a database agnostic technique for efficiently spawning replicas. Dolly is able to trigger database resynchronization operations to meet a given capacity at a given deadline while eliminating the need for database specific tools. It also allows different provisioning strategies to be implemented for private (e.g. optimizing energy usage) and public clouds (e.g. minimizing cost) while still maintaining SLA guarantees.

The key insight in Dolly is to use virtualization and the ability to clone virtual machines—a feature that is already available in compute clouds. In Dolly, each database replica runs in a separate virtual machine. Instead of relying on the traditional database mechanisms to create a new replica, Dolly clones the entire virtual machine (VM) of an existing replica, including the operating system, the database engine with all its configuration, settings and data. The cloned VM can be started on a new physical server, resulting in a new replica (which then synchronizes state with other replicas prior to processing user requests). By "black-boxing" the database state, Dolly offers a predictable replica spawning time independent of the database size and complexity. Although there are several disk-level and VM cloning techniques (commercial or research prototypes), the challenge lies in accurately quantifying the overhead and employing it for dynamic provisioning. At the heart of Dolly are intelligent models to estimate this cloning latency so that the provisioning algorithm can take the state replication cost into account when spawning replicas.

[1] Inspired by the sheep Dolly, the first mammal to be cloned successfully.

Our work on Dolly has led to the following contributions:

- We analyze the challenges of replica spawning in replicated databases and evaluate different techniques in terms of performance and manageability.
- We argue that database provisioning in the cloud requires an accurate estimation of the database backup, restore and resynchronization times. We propose a simple and effective technique based on virtual machine cloning to efficiently compute disk state replication times.
- We analyze capacity provisioning and snapshot scheduling tradeoffs and propose a new provisioning algorithm with user-defined cost functions to characterize database provisioning policies on cloud platforms. This allows the system administrator to tune the provisioning decisions to optimize resource usage of her cloud infrastructure.
- We have developed a prototype of Dolly using Sequoia [24], a commercial-grade open-source database clustering middleware, and have combined it with the OpenNebula [21] cloud manager to address provisioning in both private and public clouds. We have evaluated the effectiveness of our system by experimenting with a TPC-W [25] e-commerce workload. In our experiments, we show the ability of Dolly to properly schedule provisioning decisions to meet capacity requirements in a timely fashion while optimizing resource usage in private clouds and minimizing cost in public clouds.

The remainder of this paper is organized as follows. Section 2 introduces the necessary background on database replication and explains database replica spawning in the cloud. Section 3 discusses the core techniques for database provisioning in the cloud. Section 4 presents Dolly's implementation. We present our evaluation of spawning techniques in section 5. We perform an experimental evaluation of provisioning technique on private and public clouds in section 6. We discuss related work in section 7 and conclude in section 8.

2. SPAWNING REPLICAS IN THE CLOUD

This section introduces the necessary background on database replication (Section 2.1) and details replica spawning techniques in the cloud (Section 2.2). We then discuss the challenges of replica spawning (Section 2.3) and show how VM cloning helps when calculating replica spawning time (Section 2.4).

2.1 Database Replication

Database replication enhances scalability—by allowing replicas to collectively service a larger volume of requests—and improves availability—by allowing the system to remain operational even in the presence of replica failures. There are two primary architectures for implementing database replication: shared-disk and shared-nothing. Shared-disk requires specific hardware (i.e. a SAN) that is shared by all database replicas so that no state copy is required. Such infrastructure is usually not available in public clouds and uses product specific provisioning techniques. Dolly is not aimed at shared-disk architectures.

The alternative to expensive shared storage is the *shared-nothing architecture*. In this case, each replica has a local copy of the database content, and network communication is used to synchronize the replicas. There are two main replication strategies: master-slave and multi-master. In *master-slave*, updates are sent to a single master node, while reads are distributed among the slave nodes. Data on slave nodes might be stale and it is the responsibility of the application to check for data freshness when accessing a slave node. *Multi-master* replication enforces a serializable execution order of transactions between all replicas so that each of them executes update transactions in the same order. This way, any replica can serve any read or write request.

Replication can be implemented inside the database engine, also known as *in-core* replication, or externally to the database, commonly called *middleware-based* replication. The technique to add a new replica is similar in both environments. In both architectures, transactions are balanced among the replicas and are stored in a transactional log (also called recovery log). The middleware design usually keeps a separate transactional log for replication, whereas the in-core approach stores the information in each database's replica transactional log.

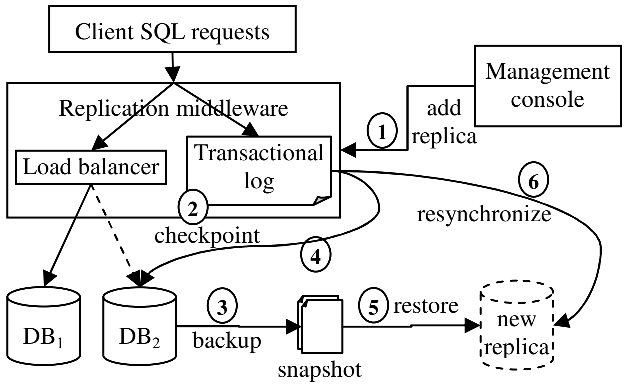

Figure 1. Procedure to spawn a replica in middleware-based replication.

Figure 1 shows the steps to spawn a replica in a middleware-based replication environment. First, a command to add a new replica is issued from the management console to the replication middleware. A checkpoint is then created in the transactional log (step 2) and a replica is temporarily taken out of the cluster to take a snapshot (also called database dump) of the database content (step 3 via DB_2). As soon as the snapshot has been taken, this replica is resynchronized by replaying the transactions written in the transactional log since the checkpoint (step 4) and it rejoins the cluster. A new replica is then started on a separate node, and the snapshot is seeded to this new replica using a restore operation (step 5). Finally, the updates that have occurred since the snapshot was taken are replayed from the transactional log (step 6) to resynchronize the new replica and bring it up-to-date with all other replicas in the system.

Conceptually, the above steps for replica creation can be classified into three key phases: (i) the *backup* phase, where database content is extracted from an existing replica and moved to a new node, (ii) the *restore* phase, where a new replica is seeded with this snapshot, and (iii) the *replay* phase, where the replica is resynchronized with others by replaying new updates from the transactional log. Dolly presently assumes multi-master middleware-based replication and is implemented on Sequoia, a commercial-grade database clustering middleware [24].

2.2 Replica Spawning via VM Cloning in Private and Public Clouds

Modern data centers and cloud platforms employ virtualization. A key management benefit is the ability to *clone* virtual machines; cloning allows a copy of the original virtual machine to be created and run on a different server. VM cloning can be exploited to efficiently spawn database replicas. Figure 2 shows how two new replicas are spawned in a private cloud. First, the virtual machine (VM) containing a database replica is stopped on machine 1 and cloned to be stored on a backup server (machine B). New replicas are spawned by cloning the VM from the backup server and starting these new VMs.

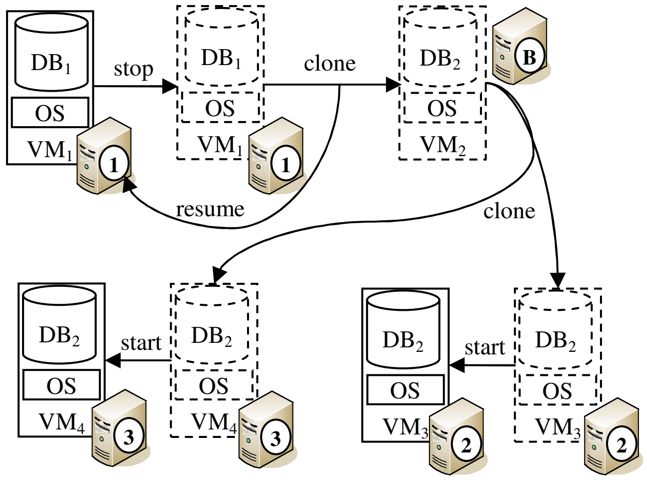

Figure 2. Replica spawning in a private cloud

To minimize the down-time of the VM being cloned, filesystem snapshots are commonly used [7]. Using this technique, VM images can be copied asynchronously, and the execution of the original VM (and the replica) can be resumed almost immediately.

Figure 3 shows how replica spawning works in a public cloud such as Amazon EC2 that provides a Network Attached Storage (NAS) service called Amazon Elastic Block Storage (or EBS). Note that EBS volumes cannot be shared by multiple instances and are therefore different from a SAN or shared disk approach. The VM disk image is stored on an EBS volume and the VM boots from this image. When the VM is stopped, the volume is detached from its running server. EBS allows snapshots of the volume to be created; doing so asynchronously replicates the volume. The volume snapshot must then be registered in EC2 in order to create new VMs. This is equivalent to storing the image to a backup server in a private cloud. When a new VM is created from an EBS snapshot, a clone of that volume is created and dedicated to the newly started instance. In our case, we assume that the database server disk state (configuration file and the data within the database) are stored on the EBS volume; thus snapshots and booting a new VM from the snapshot is an effective mechanism to replicate the shared-nothing database content and start up a new database replica.

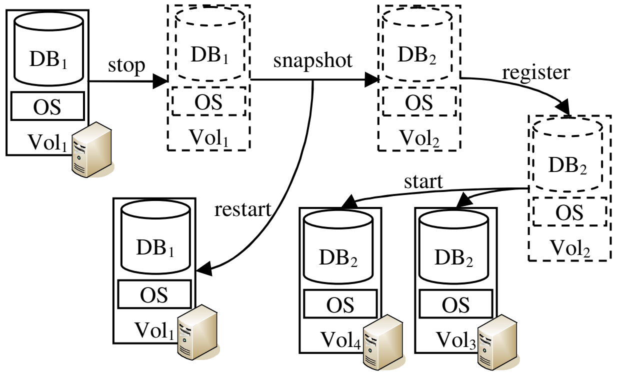

Figure 3. Replica spawning in a public cloud

2.3 Replica spawning challenges

The replica spawning procedure described on Figure 1 assumes that the replica is quickly configured and ready to receive the new database content. However, creating a new database replica involves more than just copying the database content from one machine to another. Figure 4 shows the different steps involved in the setup of a new replica. For replica spawning to be successful, all of the depicted steps must be executed flawlessly.

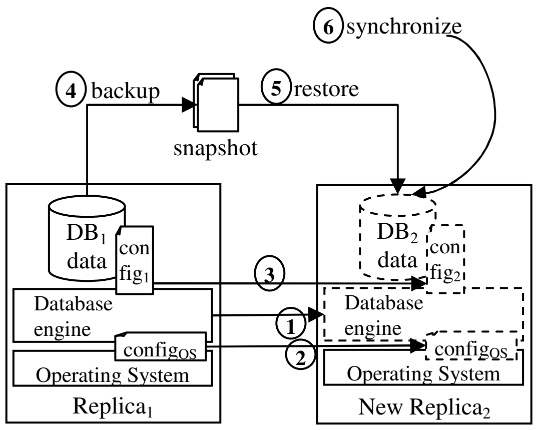

Figure 4. Steps involved in spawning a replica

First, a version of the database engine compatible with the hardware platform and operating system must be installed. The software must also be compatible with the replication software and the other database instances it has to communicate with (e.g. a master node). In step 2, the database must be configured and tuned appropriately for the hardware and the host operating system. The next step (3) is to copy the configuration parameters specific to the database instance to be replicated. This includes authentication settings, users and their respective access rights, slave identifier (for master/slave configurations), network configuration (e.g., access lists, certificate and encryptions keys for secure connections) and tool specific configurations (e.g., for backup/restore and console access).

The database content is then transferred from an existing replica by first extracting the data into files (backup on step 4) and loading it into the new replica (restore on step 5). Backup tools may not be able to capture all database objects like temporary tables, sequences, environment variables, encoding, large objects, stored procedure and trigger definitions [9]. Alternatively, filesystem-level copies can be made if the two database architectures and configurations are strictly identical.

The consequences of missing information in a snapshot transfer are many: the performance of the replica can be altered (e.g. missing index and wrong optimizer statistics), queries can fail (e.g. missing stored procedure), illegal data might be inserted (e.g. missing integrity constraint), wrong results can be generated (e.g. bad sequence number) or execution might diverge from other replicas (e.g. missing trigger or environment variable setting).

Resynchronization (step 6) is the operation that consists of replaying all the updates that happened since the snapshot was taken so that the replica can be brought up-to-date with the other nodes. This is achieved by replaying transactions from the recovery log that is kept by the replication system. A replica can be spawned from an old snapshot as long as the recovery log contains all the update transactions that the system has seen since the snapshot was taken. Similarly to the restore operation, the log replay generally must be serialized. Under a heavy write workload it is even possible that the replay mechanism does not catch up with the current workload and lags behind until the update rate decreases in the workload.

2.4 Determining replica spawning time

In general, there is a tradeoff between the time to backup/restore a database, the size of the transactional log and the amount of update transactions in the workload. For example, a new replica can be seeded with an old snapshot (e.g., a snapshot that was taken to seed a different replica), which eliminates the backup phase overhead. However, use of an older snapshot forces the system to keep a larger transactional log and also increases the time to replay updates from this log during the *replay* phase. On the other hand, taking a new snapshot for each new replica may incur significant overheads during the *backup* phase, especially if the database is large. This section analyzes this tradeoff in more detail.

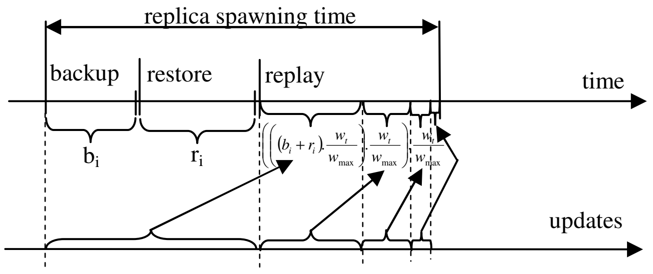
replica spawning time
backup
restore
replay
time
b_i
r_i
$\left(\left((b_i + r_i).\frac{w_t}{w_{max}}\right).\frac{w_t}{w_{max}}\right).\frac{w_t}{w_{max}}$
updates

Figure 5. Decomposition of the replica spawning time with a new snapshot

The replica spawning overhead can be analyzed using the five variables defined in Table 1.

Table 1. Replica spawning time variables.

b_i	backup time to generate VM snapshot i
r_i	time to restore/clone snapshot i on a new replica
$replay_i$	time to replay update transactions logged since snapshot i
w_t	average update transaction throughput observed at the time the new replica spawning command is issued
w_{max}	maximum update transaction throughput of the replica

When no snapshot is available, it is necessary to perform a new backup and restore, yielding an overhead of (b_i+r_i) as shown on Figure 5. The replay phase then replays all updates that have occurred during this period. We can estimate the replay time by observing the current rate of update transactions and assume that it will remain a valid approximation during the replay time. The new replica will be able to replay the requests at w_{max} speed since it does not have to execute any other transaction. Therefore, the time to replay the updates that occurred during backup/restore is $(b_i+r_i)\frac{w_t}{w_{\max}}$. Since new updates will occur during this replay, it will take an additional $\left((b_i+r_i)\frac{w_t}{w_{\max}}\right)\frac{w_t}{w_{\max}}$ to replay them. Since the system sees continuous updates, the replay will end when $\left(\frac{w_t}{w_{\max}}\right)^n \to 0$. If the system is under peak load, $w_t = w_{max}$ and the replica will never be able to catch up and it will have a lag of b_i+r_i. This is the geometric series with:

$$p=\frac{w_t}{w_{max}},\ w_t<w_{max},\ \sum_{k=0}^{\infty} p^i=\frac{1}{1-p}$$

Hence, the replica spawning time s when no snapshot is available can be estimated by the following formula:

$$s=(b_i+r_i)\frac{w_{\max}}{w_{\max}-w_t}$$

If an existing snapshot i is available, the time to spawn a new replica eliminates the backup time and is calculated as follows:

$$s=(r_i+replay_i)\frac{w_{\max}}{w_{\max}-w_t}$$

where $replay_i$ accounts for all transactions recorded when the replica spawning command is invoked.

By comparing these equations, it follows that: *it is faster to take a new snapshot j to spawn a new replica if:* b_j+r_j<r_i +$replay_i$. Any dynamic provisioning technique for replicating the database tier of the application needs to consider this key tradeoff. The VM cloning mechanism used by Dolly provides a predictable backup/restore time independent of the database size and schema complexity as shown in Table 9. Cloning only depends on the VM image size that is known and its snapshotting time can be easily predicted. $replay_i$ can be accurately predicted by recording the execution times of each update transaction and adding them up.

Since $replay_i$ can be accurately predicted, having a constant b_j and r_j, that are independent of the database size or complexity, allows Dolly to decide if b_j<$replay_i$ in which case it is faster to take a new snapshot than to use an existing one to spawn a new replica.

3. Dolly: Database provisioning in the cloud

Figure 6 gives an overview of the Dolly design. Cloud platforms come with a set of tools to manage and monitor the infrastructure. *Predictors* (Section 0) observe the behavior of the system and predict its future capacity demand. Dolly processes that information to schedule the provisioning operations using the cloud infrastructure and APIs.

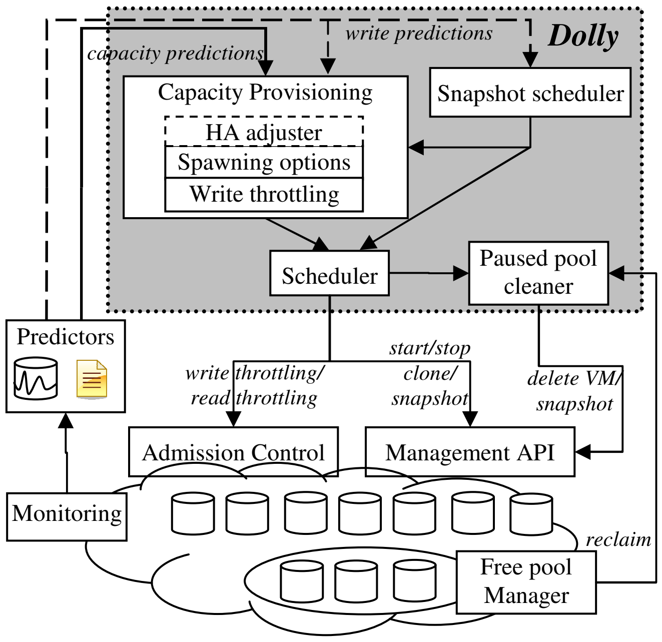

Figure 6. Dolly design overview

Dolly has four main components: capacity provisioning, snapshot scheduler, paused pool cleaner and scheduler. To meet a certain capacity at a given deadline, it is necessary to schedule *capacity provisioning* actions according to the time it takes to replicate the database state (Section 3.2). As replicas have to be spawned from a database snapshot, the *snapshot scheduler* decides when new database snapshots have to be taken (Section 3.3). Some resources (backups, paused VMs) become obsolete over time and need to be purged by the *paused pool cleaner* (Section 3.4). The scheduler orchestrates and executes the orders of the other components.

Code Sample 1. Dolly main loop algorithm pseudo-code

```
if (predictor.capacity_changes ||
    predictor.write_workload_changes) {
  do {
    schedule = capacity_provisioning(predictions)
    snapshot_schedule = snapshot_scheduling(predictions)
  } while (snapshot_schedule schedules new snapshots)
  scheduler.schedule(snapshot_schedule)
  scheduler.schedule(capacity_schedule)
}

if (time since last operation > threshold) {
  paused_pool_cleaner.release_old_paused_vms();
  paused_pool_cleaner.delete_old_snapshots();
}
```

Code Sample 1 shows a simplified pseudo-code of the main loop of the Dolly algorithm. Whenever new predictions become available, the `capacity_provisioning` algorithm is invoked to compute a new schedule to meet capacity demands. Then `snapshot_scheduling` runs to check if new snapshots could be generated (possibly from paused VMs) to make future spawning operations cheaper. If new snapshots are generated, we re-run the capacity provisioning algorithm to generate a new schedule. In the end, we obtain a schedule of snapshot and capacity provisioning actions (adding, pausing, resuming replicas) that are executed by the scheduler. Dolly also regularly triggers the `paused_pool_cleaner` to free old paused VMs and snapshots that are no longer needed.

To adapt provisioning policies to the target cloud platform, Dolly uses cost functions to allow the administrator to define which option is best if multiple strategies are available. The cost can model any metric like time, resource usage or actual resource cost as we will show in the next sections. Table 2 lists the seven cost functions used by Dolly and the definitions for each.

Table 2. Cloud platform specific cost functions used by Dolly

Cost function name	Definition
`pause_cost(VM, t)`	cost of pausing VM at time t
`spawn_cost(s, t, d)`	cost to spawn a replica from snapshot s at time t to meet deadline d
`spawn_cost(VM, t, d)`	cost to spawn a replica from a paused VM at time t to meet deadline d
`running_cost(VM,t1,t2)`	cost to run a VM from time t1 to time t2
`pause_resume_cost(VM, t1, t2)`	cost to pause a VM at time t1 and resume it at time t2
`backup_paused_cost(VM)`	cost to backup a paused VM
`backup_live_cost(VM, t)`	cost to backup an active VM at time t

Table 3 summarizes the variables used to measure the time used by the different operations used by the algorithms described in this section.

Table 3. Variables used to measure replica spawning operations**.**

rr	Time to restore and replay from the latest snapshot
br	Time to spawn from a new snapshot (backup+restore)
rs_{VM_i}	Time to resume paused VM *i*
psr	Time to pause/snapshot/resume a VM
pw	Prediction window

3.1 Capacity and workload predictors

Previous work has established how to predict replicated database capacity based on a standalone node measurement [13]. This allows forecasting performance scalability and identifying potential bottlenecks. Many models exist for workload prediction [16], [28]. Dolly does not provide any workload predictor or capacity model; it can use any existing approach and can be a platform to test new predictors or improve existing ones.

Depending on the capacity and workload predictors used, the forecast has a limited visibility in the future. Web sites with stable workloads might have accurate static weekly predictions possibly adjusted by administrators for seasonal peaks. More dynamically changing workloads can be less predictable and only sketch the demand for the next hour or so. We call *prediction window* the time between now and the latest time in the future for which the load and capacity demand can be predicted.

Figure 7 shows an example of capacity demand and write throughput of a replicated database. The prediction window slides as time goes on. Prediction windows are not necessarily of a fixed size since a predictor can dynamically change the technique it uses to forecast the load thus increasing or decreasing the prediction window size. Dolly has to schedule provisioning decisions for deadlines d_1, d_2 and d_3, where the capacity demand changes in the prediction window.

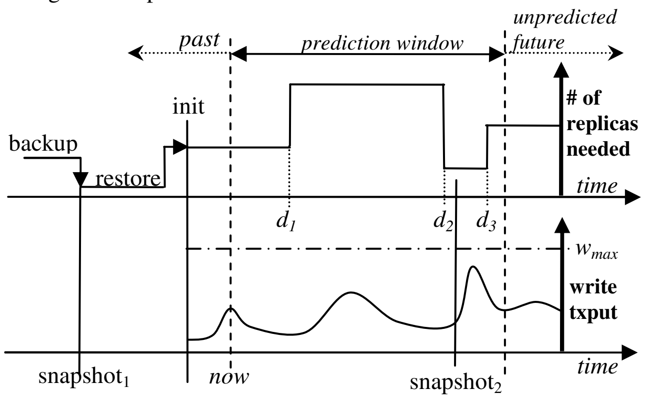

Figure 7. Example of a capacity and write workload prediction over time. Dolly provision replicas based on the forecast available in the prediction window.

3.2 Provisioning replicas

Code sample 2 gives an overview of Dolly's capacity provisioning algorithm. The provisioning algorithm scans the prediction window and looks for deadlines where changes in workload require additional capacity (such as time d_1 and d_3 on Figure 7) or less capacity (such as time d_2 on Figure 7). The algorithm handles all deadlines in sequence. In Figure 7, d_1 is handled first. Once a schedule has been found for d_1, it moves to d_2 and so on. The algorithm works in two phases for each deadline: 1) list all possible options for replica spawning or releasing and 2) sort these options according to a cost function.

3.2.1 Decreasing capacity

When the capacity requirements decrease (step 1 in pseudo-code), replicas that are no longer needed are paused. The replication middleware keeps track of the state of each stopped virtual machine replica so that it knows exactly what has to be replayed when the VM is resumed. A similar state is saved in the slave nodes for master/slave replication.

When a VM is stopped in a private cloud, its image still resides on the machine's local disk. As we might want to resume that image later, we do not return the machine to the free server pool but it is put it in a special *paused server pool*. The machine can be shutdown as long as it is in the paused pool. A machine can be reclaimed from the paused server pool by the private cloud infrastructure if the free pool is empty and additional capacity is required for other databases or tiers. In a public cloud like EC2, the computing instance is simply detached from the storage and can be re-attached later to any other instance.

The platform specific cost function, `pause_cost(VM, d)` determines the cost of pausing *VM* at time *d*. For example, in EC2 where server time is billed by the hour, if at time *d* VM_1 has just started a new billed hour and VM_2 is toward the end of its billed hour, we would have pause_cost(VM_1, d)>pause_cost(VM_2, d). On a private cloud, the administrator might prefer to switch off the hottest machines to improve cooling. If the capacity has to be reduced by *r* replicas at time *d*, the algorithm schedules the *r* replicas that have the lowest `pause_cost` for pausing.

Code sample 2. Capacity provisioning algorithm pseudo-code

```
function capacity_provisioning(deadlines[],snapshots[],
  paused_vms[], active_vms[], write_workload[])
returns schedule[] {

 foreach d in deadlines[] do {

  if (active_vms.size == d.required_capacity)
   continue // enough capacity

1 if (d.is_capacity_decrease){
   r = number of replicas to pause
   schedule += pause r VMs from actives_vms at
             time d with lowest pause_cost(vm,d)
   continue // to next deadline
  }

2 // d.is_capacity_increase
  r = number of additional replicas to spawn
  opts[] = all spawning options (initially empty)

  foreach s in snapshots[] do
3  if(now+restore&replay(s,write_workload) < d)
    opts += restore s with cost spawn_cost(s,latest,d)>

  foreach p in paused_vms[] do
4  if (now+resume&replay(p, write_workload) < d)
    opts += resume p with cost spawn_cost(p,latest,d)>

5 opts = select r cheapest options
  foreach o in opts[] {
   if (o.is_paused_vm &&
       (running_cost(p,paused_time,resume_time) <
6       pause_resume_cost(p,paused_time,resume_time)){
    schedule -= pause p // Don't pause the VM, cheaper to let it run
    continue // to next option
   }
   schedule += o
  }

  if (opts.size < r) { // could not provision all replicas in time
7  solution = capacity_provisioning() on deadline d
              with reduced write workload
   if (solution exists) {
    perform write_throttling until d
    schedule += solution
   } else { // Cannot meet capacity in time => admission control
8   schedule += restore last_snapshot now
    perform admission control until replicas are ready
   }
  }
  update active_vms for next deadline
 }
 return schedule
}
```

3.2.2 *Increasing capacity*

When an increase in capacity is predicted at deadline d (step 2 in pseudo-code), the algorithm explores all replica spawning options from snapshots and paused VMs.

In our system, the replicated database always has at least one snapshot available for creating new replicas. The first snapshot is created when initializing the system as shown on Figure 7, and snapshots are updated regularly when needed, as will be explained in section 3.3. When new replicas are spawned from a snapshot, we can predict the time it takes to bring the replica online using the formula described in section 2.3.

Dolly looks at all available snapshots that can spawn replicas in time to meet deadline d (step 3) and adds them as options. Similarly, all paused VMs that can be resumed and resynchronized in time are added as options (step 4). Each option has its own cost defined by the `spawn_cost` function. For example, on a private cloud, options using the latest start times allow unused nodes to remain switched off longer and save energy. On a public cloud such as EC2, the cost can be defined by the price the user is going to pay for the compute hours of the instance, the IOs on EBS and the monthly cost for data storage.

The cheapest options are selected to be executed (step 5). Note that if there are not enough options to provision all replicas, this means that it is not possible to spawn all replicas in time for the deadline given the current workload. We address this scenario in the next section.

A last optimization looks at all paused VMs that are going to be resumed. For each *to be resumed VM* that has not been paused yet (i.e., VMs that are scheduled to be paused sometime in the future), we compare the cost of letting the VM run—defined by `running_cost`—versus pausing and resuming it—defined by `pause_resume_cost`. In a private cloud, as long as there is enough time to pause and resume the VM (including machine shutdown and boot time), it is worth pausing the VM to save energy. In EC2, when the VM is paused the remaining unused minutes of the computing hour have been already paid for. If the VM is resumed before the end of that hour, the time will be billed twice. When `running_cost(VM, t1, t2)<pause_resume_cost(VM, t1, t2)`, it is cheaper to let the VM run and the action to pause the VM is simply cancelled (step 6).

3.2.3 *Write throttling and admission control*

If a capacity deadline cannot be met in time with the current forecast, it is possible to perform admission control on the system in multiple ways. Note that this scenario can only happen if the predictor drastically changes its predictions for the current prediction window (such as an unpredicted flash crowd).

First we assume that no writes will update the system from now on and compute the time it takes to restore and replay from the latest snapshot (*rr*), to take a new snapshot and spawn a replica from it (*br*=backup+restore) or resume from paused VMs (rs_{VM_i}). If we find that $now + min(rr, br, rs_{VM_i}, rs_{VM_j}) \le d$, this implies that there is enough time to create replicas but the write throughput is too high or too close to w_{max} for replicas to catch up in time. Doing admission control on the write throughput w_t can be used to meet the deadline as long as: $min(rr, br, rs_{VM_i}, rs_{VM_j}) \frac{w_{\max}}{w_{\max} - w_t} \le d - now$.

This translates to: $w_t \le w_{\max} - \frac{min(rr, br, rs_{VM_i}, rs_{VM_j}).w_{\max}}{d - now}$

Note that doing admission control on writes (*write throttling*), means that update transactions are going to be delayed. Depending on timeout settings, this might translate into transactions being aborted. The minimum acceptable write throughput can be set by the administrator. If the algorithm can find a solution that allows replicas to be spawned in time with write throttling (step 7 in pseudo-code), it is scheduled.

If replicas cannot be spawned in time even with write throttling, it is necessary to perform admission control on the incoming workload to prevent the system from crashing due to overload. Admission control can be performed by the replication engine by allowing only a fixed number of transactions in the system at any given time. It can also be achieved at another tier in front of the database (e.g. web tier admission control). A workload matching the current capacity has to be maintained (step 8) until additional capacity becomes available at time:

$$d - now + min(rr, br, rs_{VM_i}, rs_{VM_j})\frac{w_{\max}}{w_{\max} - w_t}$$

The administrator can set a minimum acceptable w_t and let Dolly perform admission control and schedule spawning operations accordingly.

3.3 Scheduling new database snapshots

In addition to provisioning new replicas or pausing existing ones, Dolly must deal with the problem of periodically creating new database snapshots. A newer snapshot reduces the cost of spawning a new replica in the future (since it has a more recent version of the database and will incur a lower synchronization overhead). However, creating a snapshot incurs an overhead, and Dolly must intelligently schedule their creation to balance the cost and the benefit.

Two problems have to be solved to schedule new database snapshots: *how* and *when*. *How* can either be from an already paused VM or by pausing an active VM for the time of the snapshot (see section 3.3.1). A new snapshot must be ready *when* the time to restore and replay from the previously available snapshot is greater than the prediction window (see section 3.3.2).

3.3.1 How to snapshot?

An opportunistic method to create a new snapshot is to clone VMs that have been paused. While a paused VM only captures the database state until the time it was paused, it might still be a significant improvement over the last snapshot available.

The only other option requires taking an existing replica offline for the time of the pause/snapshot/resume (psr) operation and replaying of updates that happened since the VM was paused. This means that the capacity of the system is going to be reduced by 1 replica from t_{backup} to $t_{backup} + \left(psr + replay_{t_{backup}}\right)\frac{w_{\max}}{w_{\max} - w_t}$.

If the workload prediction does not allow a replica to be temporarily disabled during that time interval, an additional replica has to be provisioned at time t_{backup} to allow taking a new snapshot. This new deadline can be added to the current capacity prediction and the capacity provisioning algorithm described in section 3.2 has to be re-executed to provision this additional replica in time.

3.3.2 When to snapshot?

If we want to provision additional replicas in time, the time to restore and replay from the latest available snapshot should never exceed the prediction window. Otherwise, when the predictor forecasts a new capacity demand increase at the end of the prediction window, there would not be enough time to spawn new replicas. This means that a new snapshot must be ready to be fully restored at time t_{switch} defined by: $r_{backup_i} + replay_{backup_i, switch} = pw$

where pw is the prediction window and $replay_{backup_i, switch} = \sum_{t=t_{backup_i}}^{t_{switch}} \frac{w_t}{w_{\max}}$

To make sure that additional replicas can be provisioned at t_{switch} using the new snapshot, the backup operation must be started the prior to time $t_{backup_{i+1}}$ so that there is enough time to backup, restore and replay a new replica at time t_{switch}. This translates to:

$$b_{backup_{i+1}} + r_{backup_{i+1}} + replay_{backup_{i+1}, switch} \le t_{switch} - t_{backup_{i+1}}$$

To guarantee that a new snapshot can be ready in time, the prediction window must be long enough so that:

$$pw \ge t_{switch} - t_{backup_{i+1}} \ge b_{backup_{i+1}} + r_{backup_{i+1}} + replay_{backup_{i+1}, switch}$$

If the prediction window is too short or write throughput is too high, admission control can be used to make sure that new snapshots can be prepared in time within the prediction window.

Code sample 3 describes the snapshot scheduling algorithm in pseudo-code. If the prediction window does not have any capacity changes, the algorithm inserts a fake capacity increase at the end of the prediction window (step 1) to make sure that at least one snapshot is available to spawn replicas in time for future demand.

The algorithm then scans the prediction window and look at each deadline where new replicas have to spawned (adding capacity only). For each deadline, it calculates the cost to spawn new replicas for 3 strategies:

1) The cost to spawn replicas from a snapshot given by `spawn_cost` (defined in section 3.2.2) for all snapshots that can be restored and replayed by the deadline (step 2).

2) For each paused VM (step 3) that can be snapshotted, restored and replayed by the deadline, the cost to take the backup from the paused VM is given by the cost function `backup_paused_cost` to which we add the cost of spawning replicas from this backup.

3) The cost of creating a backup from a live replica is given by the `backup_live_cost` function to which we add the cost of spawning replicas from this backup and the eventual cost of bringing a replica online if no idle replica is available (step 4).

Code sample 3. Snapshot scheduling algorithm pseudo-code

```
function snapshot_scheduling(deadlines[], snapshots[],
  paused_vms[],write_workload[])
returns schedule[] {

  if (deadlines.is_empty)
1  deadline += <now+prediction_window, cur_capacity+1>;

  foreach d in deadline[] do {
   if (!d.is_capacity_increase)
    continue;

   r = number of additional replicas to spawn
   min_cost = +∞;

   foreach s in snapshots[] do
    if (now+restore&replay(s,write_workload) < d)
2    min_cost = min(min_cost, r*spawn_cost(s, latest, d));

   foreach pvm in paused_vms[] do
    if (now+backup&restore&replay(pvm,write_workload) < d)
3    min_cost = min(min_cost, backup_paused_cost(pvm) +
               r * spawn_cost(pvm.snapshot, latest, d);

   new_replica&backup&restore =
    backup_live_cost(new_vm, backup_time)
4   + r * spawn_cost(new_vm.snapshot, latest, d);
   if (no idle replica at backup time)
    new_replica&backup&restore +=
     spawn_cost(last_snapshot, latest, backup_time)

   if (new_replica_and_backup_and_restore < min_cost)
5   schedule += spawn new replica + backup;
   else if (min_cost for paused VM)
6   schedule += backup paused VM;
   else if (min_cost == +∞)
   { // No snapshot available in time, force an additional replica for backup
7   deadline += latest backup time, d.capacity+1;
    invoke capacity_provisioning()
   }

   return schedule;
end
```

Next, the algorithm keeps the option that has the minimal cost for each deadline and schedules the operations accordingly (step 5 and 6). If no option is available to spawn a replica in time for a given deadline (step 7), the algorithm computes at what time a snapshot should be taken and modifies the capacity requirements to ask for one replica to be ready by that time. The capacity provisioning is then invoked to provision that replica, eventually using admission control if needed.

The capacity provisioning algorithm is re-run every time new snapshots have been scheduled (as shown on Code Sample 1) to check if a better replica spawning schedule is available. If this is the case, the old schedule is replaced by the new schedule.

3.4 Relinquishing resources

Over time, some paused VMs become obsolete and are not cost effective to be resumed. The same applies to old snapshots that need to be erased. The *paused pool cleaner* has the responsibility of releasing these resources. It is invoked at regular time intervals that can be set by the administrator (from every hour, to every day or every week). It scans each paused VM and checks the cost of resuming that VM (*spawn_cost(VM, now, pw_{end})*) and compares it to the cost of spawning a replica from the latest available snapshot (*spawn_cost(b_i, now, pw_{end})*). If the cost of resuming the VM is higher, it means that this VM will not be used anymore and it can be released.

A similar approach can be used for snapshots. All snapshots that are older than the current latest available snapshot can be released. However, the administrator might want to keep multiple older backups for recovery purposes. On a public cloud like EC2, since storage is paid for on a monthly basis, a better policy may be to retain old volumes until the end of the billing cycle.

3.5 High Availability considerations

Replicated databases are also used for their high availability features. In order to tolerate *f* faults, *f* additional replicas are needed so that at least *f*+1 replicas are running. These considerations can be easily taken into account by adjusting the capacity predictions of the predictor by adding the necessary number of replicas to tolerate the required number of faults.

Each time a node failure is detected, the provisioning algorithm must be re-executed to re-provision new replicas to replace the faulty ones. The administrator can set a hard deadline to replace the faulty replicas and therefore fix the maximum time to repair. This could trigger admission control and write throttling to meet the given deadline. A best effort replacement using paused VMs or the currently available snapshot might work for most cases.

3.6 Current limitations

Dolly assumes that all the components of the cloning operation (backup, restore, snapshot…) have a constant time which is correct for homogeneous setups with LAN interconnections. This might not be the case with heterogeneous resources or resources in different EC2 regions or clouds using WAN interconnections. The worst case scenario measurement could be taken to ensure safe scheduling, but specific optimizations for such environments are left to future work. Additional optimizations such as virtual machine migration can also be considered in these environments.

When synchronizing slave nodes in a master/slave setup, the synchronization process uses master node resources and potentially impacts its performance. We have not currently modeled this performance impact but we did not find it noticeable in our early experiments.

4. DOLLY DESIGN & IMPLEMENTATION

This section presents the design and implementation of Dolly. We first present an overview of the Dolly approach, followed by the specifics of the VM cloning technique used in Dolly. We then describe the implementation of Dolly in the Sequoia clustering middleware and the Xen virtual machine platform.

4.1 Dolly Design Overview

Dolly aims at simplifying the replica spawning process by capturing all operating system, database engine, configuration and database data as part of a single atomic operation. In Dolly, steps 1 to 5 described in Figure 4 are now a single operation: a virtual machine clone.

Figure 8 provides an overview of the complete replica spawning process in Dolly. Compared to Figure 1, note that each replica runs in a separate VM. The original spawning command is still issued to the replication middleware, which inserts a checkpoint in the transactional log and disables a backend for replication. The new replica is then created by simply cloning the virtual machine of the disabled replica. Both VMs are then restarted and resynchronized by replaying updates from the log.

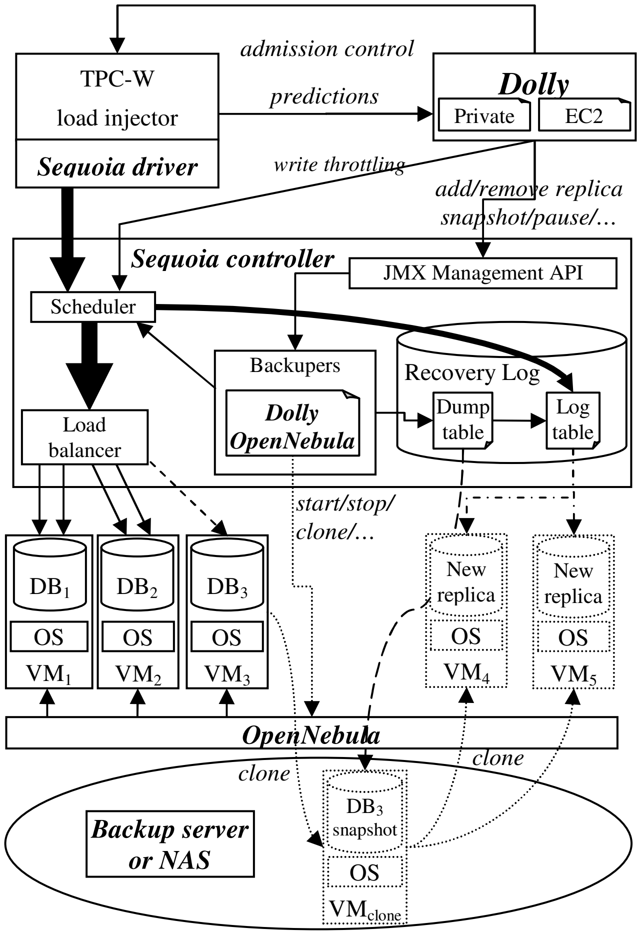

Figure 8. Overview of Dolly integration in Sequoia and OpenNebula running the TPC-W benchmark.

An extra configuration step might be required on the newly cloned VM replica (before resynchronization) in the following cases:

- Network configuration: If the IP address is statically assigned to the VM, a new address and machine name must be configured to prevent a conflict with the original VM. No such configuration is necessary if DHCP is used to obtain IP addresses.
- Replication configuration: Master-slave systems usually require each slave to have a unique id, which must be configured in the new VM.
- Security: Security is usually handled orthogonally by isolating the database cluster network within a VPN. However, when individual secure connections are required, new encryption (SSL) keys must be generated for the cloned VM.

Dolly is typically integrated in the replica spawning mechanism of the replication software and any post-restore configuration can be automated by running a script after the restore process.

4.2 VM Cloning in Dolly

Dolly supports two methods for spawning replicas: *Copy & Clone* and *Direct Clone*. In both cases, a virtual machine has to be shutdown prior being cloned. The cloned image is given a new unique id and network configuration to prevent any conflict with the original image.

Copy & Clone consists of copying a virtual machine image and cloning it on-demand each time a new replica needs to be spawned. This method is similar to the traditional database backup that generates a dump which can be re-used at will to restore new replicas.

Dolly copy & clone (Dolly c&c) spawns $\text{replica}_{\text{new}}$ from $\text{replica}_{\text{source}}$ using the following 5 steps:

1. Shutdown $\text{replica}_{\text{source}}$ VM.
2. Copy the virtual machine image files on storage.
3. Boot $\text{replica}_{\text{source}}$ VM and resynchronize it
4. Clone the stored VM image to create a new VM image on the node hosting $\text{replica}_{\text{new}}$.
5. Boot $\text{replica}_{\text{new}}$ VM and resynchronize it.

Direct Clone directly clones a virtual machine to a new replica without keeping a copy for spawning additional replicas. This approach is similar to the file system copy that directly copies files from one replica to another.

Dolly direct clone (Dolly direct) skips the VM image copy and directly transfers the image on the new replica node. The steps involved in replica spawning are:

1. Shutdown $\text{replica}_{\text{source}}$ VM.
2. Clone $\text{replica}_{\text{source}}$ image to create the virtual machine image files on the node hosting $\text{replica}_{\text{new}}$.
3. Boot $\text{replica}_{\text{source}}$ VM and resynchronize it.
4. Boot $\text{replica}_{\text{new}}$ VM and resynchronize it.

Note that both $\text{replica}_{\text{source}}$ and $\text{replica}_{\text{new}}$ can be started in parallel so that steps 3 and 4 do not have to be performed sequentially.

The benefits of using VM cloning for spawning replicas are many. The backup phase overhead is proportional to the size of the VM image and is independent of the complexity of the database schema (the overhead is indirectly linked to the database size, since the database contents are included in the VM disk image). The approach is database-agnostic and does not require any knowledge of backup-restore tools for a particular platform. The restore phase has a constant overhead—namely VM cloning and startup—and is independent of the database size. The replay phase is still necessary to resynchronize the new replica; however reducing the b_i+r_i overhead reduces the number of missed updates that must be replayed for resynchronization.

4.3 Dolly Implementation

We have implemented the concepts of Dolly in the Sequoia [24] (formerly C-JDBC [8]) database clustering middleware and integrated it with the OpenNebula cloud infrastructure manager [21]. OpenNebula works with both private and public cloud resources and offers a single API to manipulate VMs independently of the target platform. Figure 8 shows an overview of the integration of Dolly with Sequoia and OpenNebula in the context of the TPC-W benchmark.

Client applications send SQL requests to the Sequoia controller that forwards them to the underlying databases to perform replication. The SQL commands of update transactions are recorded with their execution time in a transactional log called *recovery log*. The log itself is stored in an embedded database running within the Sequoia controller. The recovery log can be replayed to synchronize new or failed replicas. Additionally, Sequoia has a replica spawning infrastructure with a pluggable *backuper* interface that interacts with the recovery log and allows for database specific implementations of backup and restore operations. Specific backupers for MySQL and PostgreSQL database engines are already provided. They invoke the native backup/restore tools provided with these databases. We have implemented two new Dolly backupers that perform a virtual machine clone operation: *Dolly copy & clone* (Dolly c&c) and *Dolly direct clone* (Dolly direct).

The cloning operation uses the *virt-clone* tool that uses the libvirt library. This allows us to remain independent of the virtualization implementation. Even though we tested our prototype with Xen, our implementation should work as is with KVM and QEMU. We use the '--nonsparse' option of the virt-clone tool to fully-allocate the guest virtual disk and ensure consistent high performance of the virtual machine. An NFS server is installed on the machine hosting the Sequoia controller. The virtual machine image copy is performed over NFS.

We have implemented a Dolly/OpenNebula backuper that interacts with OpenNebula to start/stop and clone/snapshot virtual machines to implement the backup and restore functionality. When a new backup is triggered, a pointer to the current state of the recovery log is stored with the dump metadata. When a restore operation is launched, the dump is first restored and dedicated threads then replay the recovery log (i.e. re-execute the SQL commands) from the point that was saved in the metadata. Updates are applied in a serializable order to bring the new replica in a consistent state with other replicas. The time to replay is computed by summing the recorded execution time of all queries to replay. More information about Sequoia internals and its recovery log can be found in the Sequoia documentation [24].

Dolly takes predictions directly from the TPC-W load injectors that act as oracles with perfect information. A tunable prediction window can be used from 1 minute to the entire length of the benchmark run. The provisioning actions are directly sent to the Sequoia controller through its administration interface. Dolly performs admission control directly on the load injectors but it would typically do this at the web tier level in a multi-tier setup. The write throttling is achieved by interacting with the Sequoia scheduler. We have implemented different cost functions to model our private cloud platform and the Amazon EC2 public cloud.

The private cloud cost functions detailed in pseudo-code in Table 4 optimize the time the resources are used. The longer the resources are used, the more power they use and the higher the cost. When the algorithm has to decide which VM to pause, it selects the hottest machine at that time.

Table 4. Cost function implementation for our private cloud

Cost function name	Implementation
`pause_cost(VM, t)`	`return 1/VM->machine->temp`
`spawn_cost(s, t, d)`	`return d-t`
`spawn_cost(VM, t, d)`	`return d-t`
`running_cost(VM,t1,t2)`	`return 1`
`pause_resume_cost(VM, t1, t2)`	`if (t2-t1>VM->pause+VM->resume) return 0 else return 2`
`backup_paused_cost(VM)`	`return backup_time`
`backup_live_cost(VM, t)`	`return VM->pause + backup_time + VM->resume`

Table 5 models the cost functions as the real cost the user would pay for EC2 resource usage. It includes both the compute time for server instances (charged by the hour at the *hour$* rate) and the IO cost (charged monthly per GB of storage (*EBS_storage$*) and IOs are charged per million (*EBS_io$*)). EBS snapshots are stored on S3 and are charged monthly per GB of storage (*S3_storage$*).

Table 5. Cost function implementation for Amazon EC2

Cost function name	Implementation
`pause_cost(VM, t)`	`return 60-((t-VM->start)%60)`
`spawn_cost(s, t, d)`	`comp$=(d-t)/60*hour$ io$=EBS_storage$*s->size + EBS_io$* (s->restore_io+s->replay_io) return comp$+io$`
`spawn_cost(VM, t, d)`	`comp$=(d-t)/60*hour$ io$= EBS_io$* (s->resume_io+s->replay_io) return comp$+io$`
`running_cost(VM,t1,t2)`	`(t2-t1)/60*hour$;`
`pause_resume_cost(VM, t1, t2)`	`io$= EBS_io$* (VM->pause_io+VM->resume_io) comp$=(60-(VM->stop-VM->start) %60)/60*hour$ return io$+ comp$`
`backup_paused_cost(VM)`	`return S3_storage$*s->size`
`backup_live_cost(VM, t)`	`return pause_cost(VM, t)$+ S3_storage$*s->size + (VM->stop_io+VM->start_io)* EBS_io$`

5. SPAWNING TECHNIQUE EVALUATION

This section first introduces our experimental setup and methodology. We then present our performance evaluation followed by our management evaluation.

5.1 Experimental Setup and Methodology

This section describes our experimental testbed and our experimental methodology to evaluate various replica spawning techniques.

5.1.1 Hardware and Software

We use a cluster of Pentium 4 2.8GHz machines running a CentOS 5.2 Linux distribution with a Xen-aware Linux kernel version 2.6.18-92.1.22.el5xen. We use the default packages for the MySQL and PostgreSQL databases that are included in the CentOS distribution. All machines are interconnected by a Gigabit Ethernet network. For virtualization technology, we use the popular open source Xen hypervisor (version 3.1.2). Our Dolly implementation is integrated in Sequoia 4.0 running on the Java runtime version 1.6.0_04-b12.

We experiment with 3 different benchmarks: RUBiS, TPC-W and TPC-H. RUBiS [2] is an online auction web site that is commonly used to measure distributed system performance. We use the RUBiS Virtual Appliance v1.0 from ObjectWeb [19]. TPC-W is an eCommerce benchmark from the Transaction Processing Council [27] that emulates an online bookstore. We use the ObjectWeb implementation of the TPC-W benchmark [25]. Finally, TPC-H is a decision support benchmark also from the Transaction Processing Council. We use the reference implementation from the TPC web site to generate scale 1 (1GB) and scale 10 (10GB) TPC-H databases.

5.1.2 Replica spawning techniques

We evaluate three different replica spawning techniques: backup/restore, file copy and Dolly.

Backup/restore uses the standard database tools to perform backup and restore operations. PostgreSQL backup/restore uses pg_dump and pg_restore with their default options when a SQL dump is used. A binary dump format is obtained by passing the '--format=c' to the command line. We also execute the 'vacuum analyze' command after a restore to make sure planner statistics are up-to-date. MySQL backup/restore uses the mysqldump command with the '—routines' option to generate a dump, and the mysql client to restore a dump. Databases are created and dropped using the mysqladmin tool for MySQL and pgcreate/pgdrop for PostgreSQL. We perform a standard installation of the database software on the new replica using the yum package manager.

File copy relies on database specific knowledge to locate data and configuration files. We must make sure that data is flushed to disk and consistent before performing a file copy. We simply shut down the database to make sure that all data is persisted on disk. We perform the filesystem level copy using the rsync command directly from one replica to another. The database is restarted on the new replica as soon as the copy has finished. The file copy technique is manually intensive but represents a lower bound on replica spawning time because only the bare minimum data is transferred.

Dolly performs replica spawning via virtual machine cloning. We use the two implementations described in section 4.2.

5.1.3 Methodology

We first evaluate the impact of the schema complexity on replica spawning performance. We use 3 different configurations of the same RUBiS database (same content, same number of rows, same data types). Each configuration is tested using the MyISAM table type and the InnoDB transactional type. The standard RUBiS database is referred to as 'w/ constraints'. We remove all foreign key references and corresponding indices in the 'no constraint' versions. The 'w/ constraint & index' configurations adds a full-text index on the comments and item descriptions. We report the different sizes on disk for the databases and their snapshots, as well as the performance of the different techniques to spawn a new replica for each RUBiS instance.

Second, we measure the performance of replica spawning with a new snapshot or from an existing snapshot. We assume that the database software is already installed on the new replica for the backup/restore and file copy techniques, and that the VM hypervisor is already installed when Dolly is used. Existing snapshots are assumed to be ready to restore without further processing. We do not evaluate the additional resynchronization time for new snapshots since it is strictly proportional to the length of the backup/restore operation.

Finally, we evaluate how the different spawning techniques are vulnerable to management issues that are commonly found when building new replicated databases. Manageability improves when the number of steps to perform an operation decreases.

5.2 Performance Evaluation

In this section, we evaluate the performance of the different spawning techniques. First, we look at disk space efficiency in section 5.2.1. Then we measure the influence of schema complexity on backup/restore operations in section 5.2.2. Finally, we compare the performance of replica spawning from a new snapshot and from an existing snapshot (section 5.2.3).

Table 6.Database size on disk, dump size and overall virtual machine image size for all benchmarks

Benchmark		DB size	Snapshot size	VM size
RUBiS	MyISAM no constraint	836MB	844MB	4.1GB
	MyISAM w/ constraints	1.1GB		
	MyISAM w/ constraint & index	1.2GB		
	InnoDB no constraint	1022MB		
	InnoDB w/ constraints	1.4GB		
	InnoDB w/ constraint & index	1.5GB		
TPC-W	PostgreSQL binary dump	684MB	210MB	2.1GB
	PostgreSQL sql dump		314MB	
TPC-H scale 1(GB)	PostgreSQL binary dump	1.8GB	307MB	1.1GB (OS) + 2.1GB (data)
	PostgreSQL sql dump		1.2GB	
TPC-H scale 10(GB)	PostgreSQL binary dump	12GB	2.0GB	16GB
	PostgreSQL sql dump		7.3GB	

5.2.1 Database workloads

Table 6 summarizes the database size, dump size and virtual machine size for each benchmark. The DB size column represents the amount of data that needs to be transferred by the file system copy approach. The snapshot size represents the size of the dumps generated by the backup tools. Finally, the VM size is the size of the virtual machine image on disk. The TPC-H 1GB virtual machine uses two separate partitions, one for the operating system and another for the database data.

We observe that database snapshots generated by the database backup tool generate the most compact representation. When the database schema complexity increases and when more indices are required, the database footprint on disk becomes larger. We also note that MySQL MyISAM and InnoDB storage engine have different space requirements for the same database size.

As the virtual machine has to host the operating system, database software and database content (current and future), its footprint is significantly larger. There is a tradeoff to balance for the virtual machine image size. Small images are faster to clone but will require more efforts to reconfigure when a new virtual partition needs to be added. Large images reduce the maintenance need but potentially waste disk space and inflate virtual machine cloning time.

5.2.2 Database schema complexity

We compare the performance of creating backups using the Dolly Copy & Clone technique versus the MySQL backup tools for MyISAM and InnoDB tables in the three versions of the RUBiS database presented in section 5.1.3. The results are presented in Figure 9.

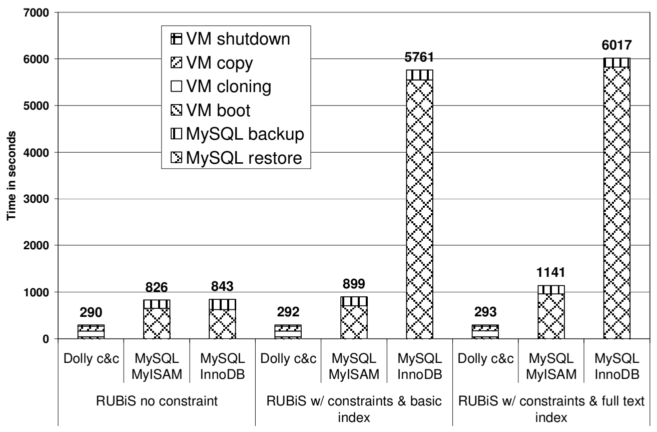

Figure 9. Time breakdown for the database snapshot transfer with Dolly (copy & clone) and MySQL with the MyISAM and InnoDB engines using the standard RUBiS benchmark initial database without constraints or index, with integrity constraints and basic indices, and with integrity constraints and full text indices.

We observe that Dolly's performance is similar for all versions of the RUBiS database. The VM shutdown and boot times are 26s and 42s, respectively. The dominant time components are the fixed cost of the VM copy to the controller (100s) and the VM cloning to the replica (varying from 123s to 125s).

MySQL MyISAM and InnoDB replica spawning performance for the database version with no constraint and index are similar at

826s and 843s, respectively. The restore operation is roughly 3 times more expensive than the backup operation. We note that the backup operation is slightly faster with MyISAM but the restore is faster with InnoDB.

The standard RUBiS database (RUBiS w/ constraints & basic index) shows a much higher overhead. While the backup times remain similar, the restore operation is much slower since it has to rebuild the indices and check integrity constraints. We have measured that the InnoDB transactional engine is 6.6 times slower than the MyISAM non-transactional engine. This effect is accentuated further when full-indices are used (RUBiS w/ constraints & full-text index). The restore time jumps to 962s for MyISAM and 5820s for InnoDB.

This experiment shows that the performance of database native backup/restore tools is affected by the database schema complexity. Dolly's blackbox approach depends only on the virtual machine image size. Dolly provides near constant performance, up to more than 20 times faster than traditional backup/restore tools. Even for a modestly sized database (about 1GB), Dolly reduces replica spawning time by a factor of at least 2.8.

5.2.3 Spawning from a new snapshot

In the following experiments, we measure the time to spawn a replica from a new snapshot. We compare Dolly with File copy and PostgreSQL backup/restore tools. Figure 10 shows the results we obtain for TPC-W. We do not evaluate the additional resynchronization time as it is strictly proportional to the length of the backup/restore operation and run against the active replica.

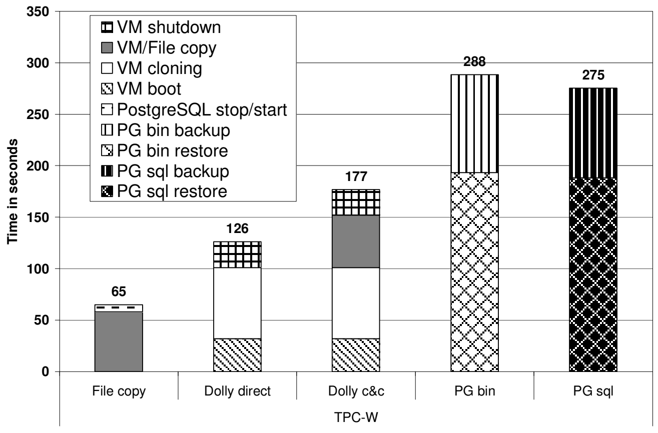

Figure 10. Time breakdown for the database snapshot transfer with Dolly (copy & clone or direct clone), File copy and PostgreSQL (binary and sql dump formats) for TPC-W.

File copy is the fastest way to replicate database content as long as the administrator knows which files need to be replicated. The database stop/restart only takes 7 seconds, and the entire spawning (assuming the database is already pre-installed on the new replica) only takes 65 seconds. Dolly has a higher overhead to shutdown and boot the virtual machine of 25 and 32 seconds, respectively. Despite its larger size, the VM image (a single file) copy over NFS is faster than the File copy replication through rsync which incurs overhead for each file to be transferred. The VM cloning, that also includes a direct file transfer over the network, takes about 69 seconds. The performance of backup/restore is similar for both binary and sql formats, with the restore operation being 3 times slower than the corresponding backup operation.

For a small database like TPC-W, the minimalistic file copy spawns a new replica in just about a minute. Dolly has an incompressible VM shutdown and boot overhead that accounts for almost a minute, doubling the time for Dolly direct compared to File copy. To achieve the equivalent of Dolly copy & clone that keeps a snapshot on the controller, two File copy operations would have to be performed (from the source replica to the controller and from the controller to the new replica). In that case, Dolly c&c reduces the overhead to about 47 seconds compared to two File copies (177s vs 130s). Dolly c&c improves over PostgreSQL backup/restore tools by 35% to 39%.

Our results with larger databases are shown in Figure 11. While File copy remains faster than Dolly direct for the smaller version of the TPC-H database, for the 10 GB TPC-H database the rsync of the larger number of data files becomes slower than the streaming of the single large VM image. The result is Dolly direct is 25% faster for the larger database. PostgreSQL's backup/restore is significantly slower than either File copy or Dolly direct. The difference between binary and sql backup/restore for TPC-H 10GB is mostly due to a much slower VACUUM ANALYZE operation when a binary dump is used for restore where this operation only takes few seconds with the sql dump.

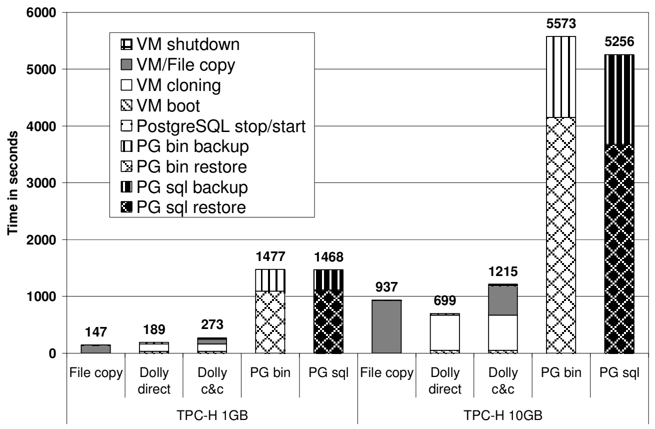

Figure 11. Time breakdown for the database snapshot transfer with Dolly (copy & clone or direct clone), File copy and PostgreSQL (binary and sql dump formats) for the TPC-H 1GB and 10GB benchmarks.

For smaller databases, Dolly has two major overheads: (i) the larger disk size of the virtual machine image that includes not only the database content but also the operating system and all software, and (ii) virtual machine shutdown and boot time. When the database size increases, these overheads become insignificant compared to the transfer of the database content. Dolly provides a much higher level of abstraction than database backup/restore tools by completely blackboxing the database, and yet still achieves a performance close to an optimized file copy that only transfers the minimal set of necessary data. Dolly has a fixed overhead of VM shutdown and boot (total 1 min), and OS image overhead (amortized with the database copy) that the minimal File copy does not. However, this additional overhead results in a maximum penalty of 1 minute in the worst-case scenario from our experiments. We believe that this is a very small overhead to pay for the gains in manageability and the elimination of errors that could lead to system corruption or downtime.

5.3 Management Evaluation

A contribution of this paper is the following summaries of the different challenges in spawning a new replica and how Dolly addresses them. The issues related here have been experienced with open source and commercial database software in production environments. Specifically, we have identified: 7 major configuration issues, 8 significant snapshot transfer issues and 4 major resynchronization issues.

Table 7. Possible management related issues in replica spawning. 'Yes' means the issue can arise for that technique.

Possible Issues	Backup/ Restore	Filesystem Copy	Dolly
Incompatible DB version	Yes	Yes	No
Database mis-configuration	Yes	Yes	No
Database not tuned	Yes	Yes	No
Missing authentication settings	Yes	Yes	No
Network configuration	Yes	Yes	Yes
Non-Unique slave id	Yes	Yes	Yes
Backup tool configuration	Yes	No	No
Expert knowledge of DB filesystem needed	No	Yes	No
Missing integrity constraint	Yes	No	No
Missing temporary tables, sequences or environment variables	Yes	Yes	No
Missing stored procedure and trigger definitions	Yes	No	No
LOBs transfer	Yes	No	No
Performance issues, concurrency limits	Yes	No	No
Non-atomic replica spawning	Yes	Yes	No
Difficult to predict replica spawning times	Yes	No	No
DB not live during resynchronization	Yes	Yes	Yes
Query restrictions during replay	Yes	Yes	Yes
Restrictions on temp objects during replay	Yes	Yes	Yes
Serialization of replay statements	Yes	Yes	Yes

Table 7 summarizes these different issues for each system. We present the details of each and conclude this section with how Dolly addresses all 15 of the configuration and snapshot issues. However, the resynchronization issues remain an open concern.

5.3.1 Configuration issues

The first step is to install and setup the database on the new replica. A compatible version of the database engine must be installed. Not only must the version be compatible with the replication software but it must also be compatible with the version used by the other replicas. The database must then be configured and tuned accordingly with the host operating system. These steps are specific to the database engine and might require extra steps for the replication configuration. Note that some tuning options might be specific to a particular database instance (e.g. size of the lock tables, buffer sizes for joins, etc.). More subtle settings such as timezone and character encoding settings can also have significant side effects on the database behavior.

Authentication settings, users and their respective access rights have to be restored. This configuration can be stored in configuration files, in the database information schema or both. A specific super user account is needed to initiate this process.

The network configuration has to be setup to accept client connections but also to communicate with the replication software and the other replicas. In a master/slave configuration a new unique slave id might have to be generated. If secure connections are used, new certificates or encryption keys have to be generated and exchanged.

Database management tools must also be able to connect to the database especially to perform operations such as backup and restore. A specific user, tool or bootstrap database might be required to connect to the database engine in order to instantiate the first database.

5.3.2 Snapshot transfer issues

The database content has to be transferred from an existing replica usually using database specific backup/restore tools. Backup tools may not be able to capture all database objects like temporary tables, sequences, environment variables, stored procedure and trigger definitions [9]. Additional challenges can arise from large objects (LOBs) or language specific extensions that need to be encoded in a neutral format.

Restore tools have to preserve data integrity which limits concurrency of operations during restore. To improve concurrency, some tools allow disabling all integrity constraints and indices during restore to speedup inserts, and only alter the tables at the end of the restore. This only applies to dumps in SQL format; snapshots in binary format have to be restored sequentially. Depending on the complexity of the database schema and the table sizes, the time to restore a database will vary which makes it very hard to predict the overall time this operation will take. Optimizer statistics have also to be recalculated after a restore operation.

Some database administrators directly backup the database files as this can be more efficient than using the backup/restore facility for large databases. Depending on the database, this task can be more or less complex. The administrator must know exactly which files to be backed up (e.g., data files, history log, redo log) for a particular instance but also where to find meta-data and configuration files, making it an error prone task. Also this becomes even more sensitive when distributed transactions are used. It is possible that some transactions are prepared but not committed at backup time and it is necessary that these states are restored properly for these transactions to complete.

The consequences of missing information in a snapshot transfer are multiple: the performance of the replica can be altered (e.g. missing index, wrong optimizer statistics), queries can fail (e.g. missing stored procedure), illegal data might be inserted (e.g. missing integrity constraint), wrong results can be generated (e.g. bad sequence number) or execution might diverge from other replicas (e.g. missing trigger or environment variable setting).

5.3.3 Resynchronization issues

Resynchronization is the operation that consists of replaying all the updates that happened since a snapshot was taken so the replica can be brought up-to-date with the other nodes. This is achieved by replaying transactions from the recovery log that is kept by the replication system. A replica can be spawned from an old snapshot as long as the recovery log contains all the update transactions that the system has seen since the snapshot was taken.

Depending on the log implementation, SQL statements are re-executed or binary logs are applied. Hybrid approaches use binary logs for small updates and SQL statements for large updates that touch many records in the database. Some systems are not capable of live resynchronization and require the system to be stopped or set to read-only to prevent additional updates to be appended to the log while recovering. Other restrictions might apply like the use of DDL statements (i.e. ALTER, CREATE, DROP…) or the use of non-persistent objects (temp tables, sequences, environment or session variables…) that cannot be logged and resynchronized properly.

Similarly to the restore operation, the log replay generally must be serialized which reduces the system's effective throughput. Under a heavy write workload it is even possible that the replay mechanism does not catch up with the current workload and lags behind until the update rate decreases in the workload.

5.3.4 Manageability

All the configuration challenges stem from the core need of recreating a compatible environment for the DBMS state at the new replica. By leveraging the virtual machine's cloning mechanism, Dolly creates an *exact* copy of the original system, DBMS and OS. The VM cloning model is a nearly perfect blackbox model that simply avoids issues due to potential differences at the copy: There are none. The exception is relatively minor changes needed for network addressing, which the cloning process automatically manages in most cases (DHCP).

The snapshot transfer issues all stem from the core need of a semantic understanding of the system state to do the transfer. The highly complex internal structures supporting DBMS systems for core functionality and additionally for optimized performance require complex tools to make minimal, semantically equivalent copies. While the database backup/restore tools provide a higher level abstraction to logically manipulate database objects, they still have limitations and make the replication process error prone. Again, the blackbox model of Dolly simply sidesteps this complexity by copying all state.

Replication related issues such as assigning a unique slave id or resynchronization problems mentioned in section 5.3.3 are independent of the cloning technique.

The benefits of replica spawning via Dolly's virtual machine cloning approach is a vastly simpler model for the administrator with essentially all the details handled by the cloning mechanism. From a manageability standpoint, Dolly provides an atomic replica spawning operation whereas other approaches require multiple steps to install, configure, copy and restore data. Following the definition of the *manageability metric*, Dolly provides a significantly higher level of manageability as it can achieve the spawning operation atomically in a comparable amount of time as file copy and much faster than backup/restore. The overhead of the Dolly approach is the transfer of extra data since, by ignoring the internal data structure semantics, DBMS state cannot be isolated.

5.4 BENEFITS AND LIMITATIONS SUMMARY

5.4.1 Administration benefits

Table 8 summarizes the features and requirements of the replica spawning techniques evaluated in this paper. Dolly's advantages over other database replication methods follow directly from the properties of cloning. Because cloning treats application state as a blackbox, Dolly can be agnostic of database specifics. This eliminates extra steps to install and configure the DBMS. Furthermore, the system state becomes a large block of data that can be streamed efficiently at near-peak component throughputs (network and disk), so Dolly replication is fast. Since no interpretation of the state is done, the time to replicate is easy to model as it is dominated by highly predictable steps: VM shutdown, state transfer, and VM boot up. Blackboxing the state into a chunk of bytes also improves safety and reliability of the copy as basic system error checking is sufficient to detect any errors. This quality essentially makes spawning a replica atomic, prior to replay of the log. In contrast, backup/restore tools have a logical layer of transfer on top of the raw byte transfer, adding a layer of complexity and more steps at which errors can occur undetected by the system and corrupting state.

Table 8. Summary of features for the three replica spawning techniques evaluated in this paper.

Feature	Backup/ Restore	Filesystem Copy	Dolly
Database specific knowledge	Medium	Very high	None
Performance	Slow	Fastest	Fast
Snapshot size	Small	DB size	VM size
Spawning time predictability	Hard	Moderate	Easy
Database installation	Moderate	Moderate	None
Database configuration	Hard	Hard	None
Missing data in transfer	Possible	Unlikely	No
Spawning atomicity	No	No	Yes
Resynchronization limitations	Yes	Yes	Yes

While Dolly provides many advantages there are some limitations. The computing cluster/cloud must be homogeneous, as Dolly's cloning, by definition, creates an exact image of the original system. For the same reason, Dolly does not replace extract, transform, and load (ETL) tools as cloning precludes any transformation of the state. Furthermore, the minimum snapshot size for Dolly is significant since the full VM state is captured. While the functional simplicity of cloning results in very efficient copying that mitigates the added delay due to the extra state, highly tuned file copies of small databases will be faster than Dolly cloning. However, so long as the system response time to spawn new replicas can be measured in a few minutes rather than seconds then Dolly cloning offers a solution that is performance competitive with file copy solutions, yet simpler and safer than conventional database backup tools.

5.4.2 Performance benefits

Like disk or filesystem replication, VM cloning is an alternative mechanism for replicating content when compared to the traditional database-specific backup-restore mechanism. In this section, we summarize the copy overheads of the two approaches.

Table 9 summarizes the time to copy various databases using the database native backup/restore tool (e.g. mysqldump, pg_dump) versus VM cloning. The RUBiS benchmark database [2] is tested with 3 configurations on MySQL using the InnoDB engine: without constraint or index (-c-i), with integrity constraints and basic indexes (+c+bi) and with constraints and full text indexes (+c+fi). TPC-W and TPC-H [27] databases are stored in a PostgreSQL RDBMS. We also experiment with two virtual machine image sizes (4 and 16GB) where we store both the operating system and the database within its content.

Table 9. Backup/restore and VM cloning time in seconds for various standard benchmark databases.

Database	DB size on disk	DB Backup Restore	Dolly 4GB VM cloning	Dolly 16GB VM cloning
RUBiS –c–i	1022MB	843s	281s	899s
RUBiS +c+bi	1.4GB	5761s	282s	900s
RUBiS +c+fi	1.5GB	6017s	280s	900s
TPC-W	684MB	288s	275s	905s
TPC-H 1GB	1.8GB	1477s	271s	918s
TPC-H 10GB	12GB	5573s	n/a	911s

Indexes significantly increase the database footprint on disk. We observe from the RUBiS results that integrity constraints checks as well as index building can increase database backup/restore time by a factor of more than 7 for the exact same database content. Not only do the database schema and backup tool configurations affect timings, different database engines yield very different results for databases with a similar size on disk. We observe that large or complex databases can take more than 1 hour to replicate.

In contrast, VM cloning performs a filesystem level copy without interpreting database objects, thus it offers a constant time regardless of the database complexity or engine used. The time only depends on the VM image size on disk (280s for a 4GB image and about 900s for a 16GB image). Consequently, since the VM disk size is fixed a priori, VM cloning makes it easy to predict database backup/restore time incurred when spawning a new replica—a crucial pre-requisite for database provisioning.

6. PROVISINONG EVALUATION

This section first introduces the cloud platforms used for our experiments. We then present our performance evaluation.

6.1 Cloud Platforms

We use a private cloud composed of a cluster of Pentium 4 2.8GHz machines. Each machine is running a CentOS 5.4 Linux distribution with a Linux kernel version 2.6.18-128.1.10.el5xen, the Xen 3.3 hypervisor and MySQL v5.0.45. All machines are interconnected by a Gigabit Ethernet network.

We use Amazon EC2 as our public cloud. EC2 instances are created from EBS volumes. We use standard large on-demand EC2 instances in our experiments. Each EC2 instance has CloudWatch running on it to monitor the number of writes. The price of our EC2 instance with CloudWatch is $0.355 per hour. The price of an EBS volume is $0.10 per allocated GB of data per month. The cost of doing I/O requests to an EBS volume is $0.10 per million I/O requests. There is a cost of $0.15 per GB per month associated with the storage of EBS volume snapshots.

Table 10. Operation timings in seconds for a TPC-W benchmark virtual machine on our private cloud and EC2.

Operation	Private Cloud	Public Cloud (EC2)
start VM	142s	220s
pause VM	26s	30s
resume VM	42s	30s
backup (stop/clone)	150s	320s
restore (clone/start)	132s	220s
w_{max}	149 writes/sec	197 writes/sec
Avg IOs per write	15	13

Our Dolly implementation is integrated with the Open-Nebula cloud management framework v1.4 and Sequoia 4.0 running on the Java runtime version 1.6.0_04-b12. We build a 4GB VM image of the TPC-W benchmark for both cloud platforms. We report our measurements of the various VM management and cloning operations in Table 10. We measure the maximum write throughput of a single replica (w_{max}) obtained by running only write transactions of the TPC-W workload on a standalone database. The average number of IOs per write transaction is calculated by running iostat before and after the w_{max} run.

6.2 Workload Description

We experiment with the TPC-W benchmark. TPC-W is an eCommerce benchmark from the Transaction Processing Council [27] that emulates an online bookstore. We use the ObjectWeb implementation of the TPC-W benchmark [25]. The setup is similar to the one depicted in Figure 8 with load injectors providing a 2 hour prediction window. The web tier (not shown on Figure 8) is statically provisioned with enough servers for the length of the experiment.

We have generated a custom mix of interactions to create the workload depicted at the top of Figure 12. We generate a read-only request mix by using the TPC-W browsing mix workload and removing its few write interactions. We use httperf to create the desired number of clients that send these read-only interactions. The write interactions are generated using the customer registration servlet of TPCW. Another set of httperf clients generate these write-only interactions.

We use the model described in [13] to determine the capacity requirements shown in Figure 12. The initial capacity demand at t=0 is 4 replicas (middle graph) and the write throughput is 20% of the maximum write throughput (bottom graph). After 10 minutes the number of replicas needed decreases from 4 to 3. We denote this deadline by d_1. The number of replicas needed decreases further from 3 to 2 at d_2=20 minutes. The capacity demand increases sharply from 2 to 5 replicas at d_3=80 minutes, then drops to 2 at d_4=90 minutes and increases up to 6 replicas at d_5=100 minutes. The number of writes remains constant to 0.2 times the maximum write throughput for one hour with a 10 minute read-only workload starting at d_2. After that hour, the write throughput is 0 until d_3 with a write surge at 50% of the maximum write throughput. The write peak continues for 10 minutes and the write throughput drops to 0 at d_4.

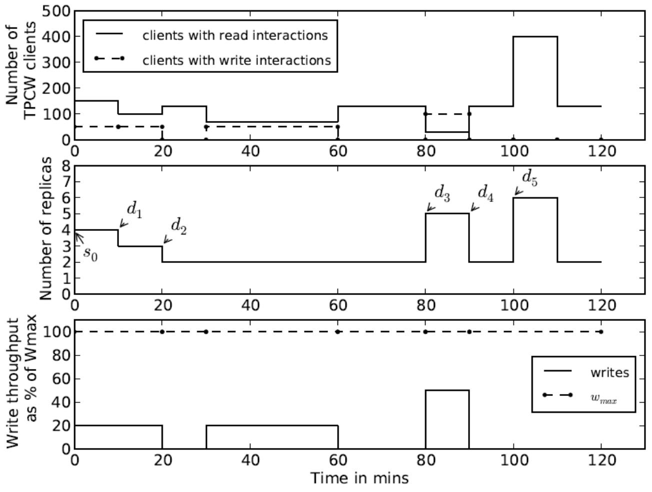

Figure 12. TPC-W workload, predicted capacity requirements and write workload.

A snapshot s_0 is available at time t_0. The 4 initial VMs v_1, v_2, v_3 and v_4 are started at t_0-10, t_0-20, t_0-30 and t_0-40 minutes, respectively. In the private cloud, machines are stacked in a rack with cooler temperatures at the bottom of the rack. VMs with lower numbers (e.g. v_1) are assumed to run on cooler machines.

6.3 Provisioning Schemes Evaluated

We evaluate Dolly's performance by comparing it with two other traditional provisioning schemes: *reactive provisioning* and *overprovisioning*. Both these schemes have the same behavior on both public and private cloud.

Reactive provisioning does not use any prediction and just reacts to the current capacity demand. Under this scheme VMs are paused when the demand decreases and are destroyed when a newer snapshot becomes available; also database snapshots are taken at regular intervals. We use intervals of 15 minutes (Reactive15m), 1 hour (Reactive1h) and 2 hours (Reactive2h), generating 7, 1 and 0 snapshots respectively during the experiment.

The overprovisioning configuration (Overpro6) uses a constant set of 6 nodes. Just like reactive provisioning, snapshots are generated periodically in this scheme. We choose to only generate 1 snapshot during the experiment.

We invoke the Dolly provisioning scheme on the public (Pub) and private (Pvt) cloud with three different prediction windows of lengths 10 minutes (Dolly10mPub and Dolly10mPvt), 30 minutes (Dolly30mPub and Dolly30mPvt) and 2 hours (Dolly2hPub and Dolly2hPvt).

In the following sections, we invoke each provisioning scheme in the private and public clouds with the workload and initial conditions defined in the previous section. We use two metrics to capture the performance of each provisioning scheme. The first metric is the cost of the provisioning scheme. The cost in the private cloud represents the cumulative machine uptime (6 machines up for 5 minutes accounts for 30 minutes) while the cost in the public cloud (Amazon EC2) is the real cost in $USD. The lesser the cost, the better the performance of the provisioning scheme. The second metric used is *missing replica minute* (MRM) that measures capacity underprovisioning (i.e. SLA violations). 1 MRM corresponds to a missing capacity of 1 replica for 1 minute (5 replicas missing for 2 minutes accounts for 10MRM). The lesser the MRM of a provisioning scheme, the better the performance of the provisioning scheme.

6.4 OverProvisioning

We evaluate the performance of the *overprovisioning* technique. This scheme always provisions six database replicas for the system. A snapshot is taken at t=60 minutes. One of the six replicas is used to generate this snapshot and the replica is therefore unavailable for processing the workload during the time the backup is being taken i.e., for 2m30s.

As 6 nodes are used for the 2 hours of the experiment, the total cost is 720 minutes for the private cloud. The system is never underprovisioned and therefore the MRM is 0. The EC2 cost is dominated by machine rental cost and totals $8.39.

6.5 Reactive Provisioning

In this section we describe the provisioning decisions made by reactive provisioning with database snapshots taken at intervals of 15 minutes, 1 hour and 2 hours. This provisioning scheme takes the same decisions on both private and public clouds, so we only illustrate the decisions made by reactive provisioning on the private cloud.

6.5.1 Reactive provisioning with 15 min snapshots

Reactive provisioning responds to the decrease in demand at d_1 and d_2 by pausing VMs v_4 and v_3. The provisioning scheme needs to generate a new database snapshot at t=15 minutes. To generate this snapshot, it spawns a new database replica from snapshot s_0. The replica is spawned by restoring from s_0 and then replaying the writes, that takes a total of 6m12s. A new snapshot s_1 is thus generated at 21m12sec. The next snapshot is required at t=30 minutes. A new replica is spawned from s_1 to generate this snapshot. Spawning this replica involves restoring from s_1 and then replaying the writes, that takes a total of 2m45s. A new snapshot s_2 is generated from this replica. A new replica spawning is started at t=45 minutes to generate the next snapshot. Spawning this replica from s_2 takes a total time 3m26sec, that includes time to restore from s_2 and then replay the writes. A new snapshot s_3 is thus generated at t=48m26s. Again, at t=60 minutes, a new replica is spawned to generate a new snapshot. A new replica is spawned by restoring from s_3 and replaying, that takes 4m30s, and then a new snapshot s_4 is taken from this replica. Next, at t=75m, a new snapshot is needed. The last snapshot s_4 is uses to spawn a new replica in 2m12s by just restoring. A new snapshot s_5 is taken at 77m12s. The provisioning scheme needs to spawn three replicas to meet deadline d_3=80 minutes. It spawns these three replicas by restoring and replaying from snapshot s_5; the replica spawning takes 4m24s during which the system remains underprovisioned by three replicas. When the demand decreases at d_4=90 minutes, the reactive provisioning scheme pauses the three replicas v_1, v_2, v_5. A new snapshot is required at t=90 minutes. A replica spawning operation is started at t=90 minutes by restoring from s_5 and then replaying the writes. The replica is ready at t=97 minutes 12 seconds at which point a new snapshot s_6 is generated. For deadline d_5=100 minutes, four replicas need to be spawned. Three replicas are spawned by resuming the VMs v_1, v_2, v_5. The fourth replica is spawned by restoring from snapshot s_5 that takes 2m12s during which the system is underprovisioned by one replica.

6.5.2 Reactive provisioning with 1 hour snapshots

Just like the previous provisioning scheme this provisioning scheme pauses VMs v_4 and v_3 for deadlines d_1 and d_2. A new snapshot is required at t=60 minutes, so a new replica needs to be spawned. The replica spawning operation is started by restoring from snapshot s_1 and replaying, that takes 12m12s. A new replica is available at t=72 minutes 12 seconds and a new snapshot s_2 is generated by taking its backup. For deadline d_3=80 minutes three replicas are spawned by restoring and replaying from this snapshot s_2. The new replicas become available at t=84 minutes 48 seconds and thus the system remains underprovisioned for 4m48s. Replicas v_1, v_2, v_5 are paused at d_4=90 minutes when the demand decreases. To meet the deadline d_5, three replicas are spawned by resuming v_1, v_2, v_5 and the fourth replica is spawned by restoring from s_2 and replaying that takes 7m12s during which the system is underprovisioned by one replica.

6.5.3 Reactive provisioning with 2 hour snapshots

We now explore the provisioning decisions taken when the reactive provisioning scheme is used with snapshots taken every 2 hours. This scheme pauses VMs v_4 and v_3 for deadlines d_1 and d_2. For deadline d_3, the three replicas are spawned by restoring from s_0 and replaying that takes 17m12s. This scheme remains underprovisioned by three replicas for the entire 10 minutes duration between t=80m and t=90m. The three replicas are paused at t=97 minutes 12 seconds. These VMs are then resumed to meet the deadline d_5. The fourth replica needed to meet deadline d_5 is spawned by restoring from s_0 and replaying, that takes 17m12s. Thus, the system is underprovisioned by one replica from t=90 minutes to t=100 minutes.

6.6 Dolly

In this section we describe the provisioning decisions made by Dolly with different prediction windows (10 minutes, 30 minutes and 2 hours) in the private cloud and public cloud.

6.6.1 Dolly 10 min prediction window

We consider a sliding prediction window of 10 minutes. As Section 3.3.2 describes, the snapshot scheduling algorithm continuously adds a fake deadline of spawning one replica at the end of the prediction window. When this deadline cannot be fulfilled by restoring and replaying from the latest snapshot, the snapshot scheduling algorithm schedules a new snapshot. Thus, with a 10 minute prediction window Dolly produces multiple snapshots during the experiment.

<u>Private Cloud:</u>

For deadlines d_1 and d_2 where the demand decreases Dolly pauses v_4 and v_3 just like in the earlier case. When the start of the prediction window reaches t=29 minutes, the snapshot scheduling algorithm adds a deadline of spawning one replica at the end of the prediction window. The snapshot scheduling algorithm realizes that this deadline of spawning a new replica at t=39 minutes cannot be fulfilled by restoring and replaying from the existing backup s_0. So, it needs to create a new snapshot from which it can spawn a new replica. The only option to spawn a new replica for taking this new snapshot is by unpausing the paused VM v_3 at t=32m and then replaying the writes. We then take a new snapshot s_1 at t=32m. Dolly pauses the VM v_3 after the snapshot operation is finished. Dolly could also have used unpaused v_4 to spawn the replica needed for taking this snapshot, but since the cost of spawning from v_4 is more than the cost of spawning from v_3, it chooses to unpause v_3. When the start of the prediction window reaches t=70 minutes, Dolly needs to spawn three replicas for deadline d_3. The capacity provisioning algorithm evaluates the costs of the three options available; spawning from the latest backup s_1 has cost 8m, the cost of spawning by resuming v_4 and replaying is 8m42s and the cost of spawning by resuming v_3 and replaying is 6m. The capacity provisioning algorithm therefore schedules spawning a replica by resuming v_3 at d_3-6m and the replaying. It schedules spawning the remaining two replicas by restoring from snapshot s_1 at d_3-7m34s and then replaying. The snapshot scheduling algorithm explores if it is possible to achieve deadline d_3 by creating a new snapshot and spawning from it. Since the deadline is only 10 minutes away, the snapshot scheduling algorithm finds it impossible to take a new snapshot and spawn from it in such a small time. When the start of the prediction window reaches t=74 minutes, the snapshot scheduling algorithm again realizes that it needs to create a new snapshot to meet the deadline at the end of the prediction window i.e. at t=84 minutes. To create this new snapshot, the snapshot scheduling algorithm explores the options of spawning a new replica. The only option to spawn a new replica is by restoring from s_1 and replaying. Thus, the snapshot scheduling algorithm schedules restoring a new replica from s_1 at t=74 minutes and replaying. The replicas become available for taking a new snapshot at t=82 minutes when we take a new snapshot s_2. When the demand decreases at d_4 Dolly decides to pause VMs v_1, v_2 and v_5. When the start of the prediction window reaches t=90 minutes, Dolly needs to spawn four replicas for the deadline d_5. Just like in the case of Dolly with 2 hour prediction window, in this case Dolly decides to unpause v_1, v_2, v_5 to spawn three replicas at d_5-42s and schedules a new backup from one of the paused VMs at t=d_5-4m42s and restore from it to create a replica.

<u>Public Cloud:</u>

Dolly makes similar decisions in public cloud with a 10 minute prediction as in the case of the private cloud. Dolly pauses v_4 and v_3 at deadlines d_1 and d_2. When the start of the prediction window reaches t=21 minutes 40 seconds, Dolly's snapshot scheduling algorithm adds a deadline of spawning a replica at the end of the prediction window i.e. at t=31 minutes 40 seconds. It realizes that this deadline cannot be fulfilled by spawning this replica from the latest snapshot s_0 and new snapshot needs to be taken. To take a new snapshot a replica needs to be spawned whose backup can be taken. Note that the time to take a backup and then restore in the public cloud totals 9m. Thus the only option for generating a new snapshot and spawning a replica at the end of a 10 minute prediction window is to resume the paused VM v_3 at t=21 minutes 40 second, replay and then take a new backup s_1. Similarly, when the start of the prediction window reaches t=41 minutes 40 seconds Dolly's snapshot scheduling algorithm adds a new deadline of spawning a new replica at t=51 minutes 40 seconds. This deadline cannot be fulfilled by spawning a replica from the latest snapshot s_1, so the snapshot scheduling algorithm tries to generate a new snapshot by spawning a new replica and taking its backup. Because of the small prediction window, the only option available with the snapshot scheduling algorithm is to resume v_3 at t=41 minutes 40 seconds, replay and then take its backup, just like before. Dolly thus generates a new snapshot s_2 at t=45 miniutes 12 seconds. When the start of the prediction window reaches t=70 minutes, Dolly needs to spawn three replicas for deadline d_3. The capacity provisioning algorithm evaluates the cost of the three options Dolly has: restoring from s_2 and replaying costs \$0.48, resuming v_3 and replaying costs \$0.46, and resuming v_4 and replaying costs \$0.57. The capacity provisioning algorithm therefore schedules spawning a replica by resuming v_3 at d_3-3m27s and replaying. It schedules restoring from s_2 at d_3-6m37s and then replaying to spawn the remaining two replicas. The snapshot scheduling algorithm explores options of spawning replicas by generating a new snapshot, but the small prediction window precludes spawning new replicas by generating a new snapshot. When the start of the prediction window reaches t=76 minutes 45 seconds, the snapshot scheduling algorithm again adds a new deadline of spawning a replica at t=86 minutes 45 seconds. The snapshot scheduling algorithm schedules generating a new snapshot since the last snapshot s_2 is too old to be used for spawning this replica. To generate this new snapshot the snapshot scheduling algorithm needs to spawn a new replica and take its backup; the only option to generate a new replica in time is to restore from s_2 and replay. Thus Dolly immediately starts restoring from s_2 at t=76 minutes 45 seconds. A new snapshot s_3 is generated at t=86 minutes 45 seconds after the restore and replay have finished. Dolly pauses v_1, v_2, v_5 when the demand decreases at d_4. When the start of the prediction window reaches t=90 minutes, Dolly needs to provision four replicas to meet the deadline d_5 at the end of the prediction window. The capacity provisioning algorithm compares the two options available for spawning these replicas: resuming the paused VMs costs \$0.40, restoring from s_3 and replaying costs \$0.45. The snapshot scheduling algorithm evaluates the cost of creating a new snapshot from the paused VM and then spawning from that snapshot, the cost of this option is \$0.43. Dolly therefore schedules unpausing the paused VMs at d_5-30s and spawning three replicas. For the fourth replica, Dolly schedules resuming a paused VM at t=d_5-9m30s, taking a new snapshot s_4 and then restoring from this snapshot.

6.6.2 *Dolly 2 hours prediction window*

With a 2 hour prediction window, Dolly is able to plan for the entire duration of the experiment. However, a 30 minute prediction window leads to exactly the same decisions and results.

As Section 3 describes, Dolly repeatedly invokes the capacity provisioning algorithm followed by the snapshot scheduling algorithm until no new snapshots are scheduled. First, the capacity provisioning algorithm is executed.

Phase 1: Capacity provisioning algorithm

The provisioning decisions taken by the capacity provisioning algorithm in this first phase are summarized in Table 11. The algorithm looks at each deadline sequentially and chooses the optimal options to meet that deadline. For the first deadlines d_1 and d_2, the algorithm has to decrease the capacity of the system by 1. The algorithm uses `pause_cost` to determine which VM to pause. In the private cloud, it decides to pause the hottest VMs (v_3 and v_4). For EC2, v_3 and v_4 are chosen since they have used most time of their billed hour.

Three replicas have to be provisioned at d_3. On both platforms, resuming the paused VMs is cheaper than spawning replicas from snapshot s_0. Therefore, 2 replicas are provisioned by resuming v_3 and v_4, and 1 additional replica is spawned by restoring s_0. The scheduling of the operations is done according to the timing of the operations for each platform that are summarized in Table 10.

Table 11. Actions scheduled by the capacity provisioning algorithm (phase 1) for each cloud platform

Deadline	Private Cloud	EC2
d_1	Pause v_4	
d_2	Pause v_3	
d_3	Resume v_3 @ d_3-7min Resume v_4 @ d_4-9min Spawn v_5 from s_0 @ d_3-13min	
d_4	Pause v_1,v_2,v_3	-
d_5	Resume v_1,v_2,v_3 @ d_5-6min	-
	Spawn v_6 from s_0 @ d_5-18min	

The algorithm uses `pause_cost` to determine which 3 replicas to pause at d_4. These paused VMs are the cheapest option to provision 3 of the 4 replicas needed by d_5. The 4th replica is spawned from s_0. The algorithm uses the `pause_resume_cost` function to determine if the decision to pause the VMs should be changed or not. In the private cloud, there is enough time to pause and resume the VMs so it makes sense to pause them. In EC2, `pause_resume_cost` is dominated by the instance cost. It costs \$0.059 to let the VM runs for 10 minutes compared to \$0.296 for the 50 minutes wasted if the VMs are paused. So the algorithm decides to not pause the VMs for EC2.

Phase 2: Snapshot scheduling algorithm

Now, the snapshot scheduling algorithm is executed to check if creating new snapshots after s_0 could yield cheaper replica spawning costs. The decisions scheduled are shown in Table 12. The snapshot scheduling algorithm looks at each deadline where new replicas have to be spawned (d_3 and d_5) and decides the best strategy to take a snapshot for that deadline. For d_3, the algorithm uses `spawn_cost` to determine the cost of spawning 3 replicas from the initial snapshot s_0. It uses `backup_paused_cost` to determine the cost of taking a backup of each paused VM. Finally, it calculates the cost of taking a backup of a live replica using `backup_live_cost`.

In the private cloud, the cost to spawn 3 replicas from s_0 is 36min36s (3*(132s+10m)), spawning from v_4 is 33m06s (150s+ 3*(132s+8m)), spawning from v_3 is 31m06s (150s+3*(132s+6m)). Finally the total cost of taking a snapshot from a live replica is spawning from s_0 (12m12s) followed by backup (2m56s) and 3 restore/replay (3*132s=6m36s), that is a total of 21m44s. This last option has the smallest cost. The algorithm then schedules the spawning of a new replica at time d_3-18min that leaves enough time for spawning, snapshot, restore and replay by d_3.

In EC2, the cost to spawn 3 replicas from s_0 is \$1.89 (3*(running cost (\$0.355*0.22) + EBS volume cost (\$0.10*4) and EBS IO cost (\$0.10*1.54))), spawning from a snapshot of v_4 is \$2.34 (snapshot storage cost (4*\$0.15) + 3*(running cost (\$0.355*0.16) + EBS volume cost (4*\$0.10) and EBS IO cost (\$0.10*1.3))), spawning from a snapshot of v_3 is \$2.22 and spawning from a snapshot of a live replica is \$2.50 (\$0.84 to spawn a new replica from s_0 + \$1.26 to spawn 3 replicas from the snapshot). As it is cheaper to spawn replicas from s_0, no new snapshot is scheduled.

The second deadline where capacity needs to be increased is d_5. We need to spawn 4 replicas for the private cloud but only 1 for EC2 where the VMs were not paused. The same cost functions are evaluated again and the private cloud schedules a new snapshot but this time from a VM paused at d_4. In EC2, the storage cost of EBS snapshots and volumes still overcomes the cost of replaying IOs. Therefore, as restoring from s_0 still allows spawning replicas in time and it is still the cheapest solution, the algorithm decides to not schedule any new snapshot. As no new snapshot is scheduled, the algorithm terminates here for EC2.

Table 12. Scheduling decisions of the snapshot scheduling algorithm (phase 2) for each cloud platform

Deadline	Private Cloud	EC2
d_3	Spawn replica v_5 at d_3-18m + snapshot s_1	-
d_5	Snapshot s_2 from paused v_1@ d_4+1min	-

Next iteration

In the second iteration, the capacity provisioning algorithm is invoked again for our private cloud. As more snapshots are available, new decisions are scheduled as shown in Table 13. As it is cheaper to spawn replicas from s_1 than to resume v_3 and v_4, these 2 VMs will never be resumed and get cleaned up by the paused pool cleaner when it is invoked. For d_3, two replicas are spawned from s_1 and one is provisioned by resuming v_5, the VM that was used to take the snapshot. At d_5, the 3 paused VMs are resumed and an extra VM is spawned from the new snapshot s_2.

Table 13. Actions scheduled by the capacity provisioning algorithm (phase 3) for the private cloud

Deadline	Private Cloud
d_1	Pause v_4
d_2	Pause v_3
d_3	Resume v_5 @ d_3-1min Spawn v_6,v_7 from s_1 @ d_3-3min
d_4	Pause v_1,v_2,v_5
d_5	Resume v_1,v_2,v_5@ d_5-6min Spawn v_8 from s_2 @ d_5-3min

The snapshot scheduling algorithm is invoked again. No new snapshot is scheduled as it is not possible to find options with a cheaper cost than what is currently available with s_1 and s_2.

Final Phase: Scheduling

Since the snapshot scheduling algorithm does not create any new snapshots the loop terminates and the schedules are sent to the scheduler for execution. Figure 13 shows the final provisioning decisions made by Dolly for EC2 and our private cloud platform.

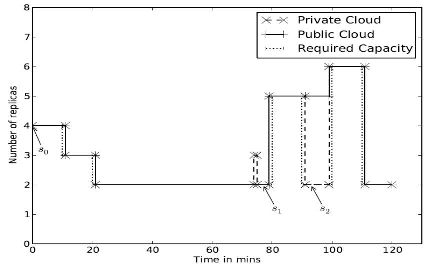

Figure 13. Final provisioning decisions taken by Dolly with a 30 minutes or 2 hour prediction window for our private cloud and Amazon EC2 (public cloud).

Both schedules are correct and meet the capacity requirements in time. The cost functions have optimized the schedules for the minimum cost on EC2 and for lower energy consumption on our private cloud.

6.7 Summary

We summarize the performance of the various provisioning schemes evaluated in the preceding sections in this section. Figure 14 shows the provisioning decisions taken by each algorithm as explained in the previous section. The performance of the different algorithms as indicated by the two performance metrics is summarized in Table 14.

The results show that reactive provisioning is not able to properly provision the system with missing capacity ranging from 23.2 to 44.2 missing replica minute. Snapshotting more often reduces the time to spawn new replicas by restore and replay but capacity is missing during the spawning operations.

Overprovisioning (Overpro6) always provides an adequate capacity but at a significantly larger cost on each cloud platform. In contrast, Dolly uses much less resources while still providing the required capacity. A 10 minute prediction window (Dolly10m) requires more snapshots to be able to react to any new capacity demand at the end of the short prediction window. A 30 minute prediction window (Dolly30m) is enough to provide an optimal provisioning using less than half of the resources of the overprovisioned configuration.

When reactive provisioning is used, additional capacity is used to spawn a new replica from the latest snapshot so that a new snapshot can be generated. When capacity needs to be increased, the system remains underprovisioned during the time replicas are spawned. The older the snapshot the longer it takes to spawn new replicas. In the Reactive2h case, replicas spawning starting at t=80 completes only 17 minutes later, leaving the system with only 2 available replicas to serve requests during the first peak period.

The Overpro6 configuration constantly provides 6 replicas except for when the snapshot is generated where a node is briefly paused. The large shaded area shows the amount of wasted resources.

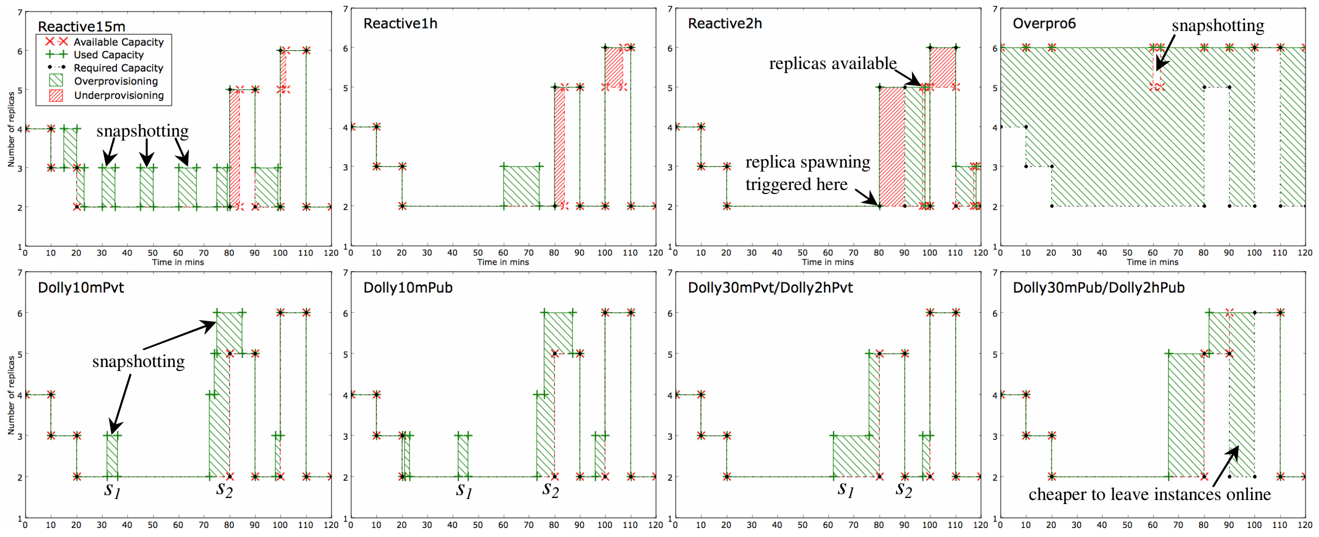

Figure 14. Capacity made available by each provisioning algorithm compared to the required capacity and the total capacity actually used.

Dolly with a 10 minute prediction window (Dolly10m) behaves similarly on both cloud platforms. As the prevision window slides the time to restore and replay from the latest snapshot exceeds the prediction window size. This is why Dolly spawns new replicas to generate new snapshots at deadlines s_1 and s_2. While new replicas are spawned from s_1 during the first capacity increase, the write spike quickly triggers an additional replica to generate s_2. Four replicas are paused at the end of the first peak and resumed for the second peak (no replay time since no write occurred during that paused time). An additional replica is quickly spawned from s_2.

With a 30 minute or longer prevision window (Dolly30m and Dolly2h), decisions change between the private and the public cloud according to the cost functions. While less machine time is used on the private cloud by generating new snapshots from an additional replica online (s_1) or from a paused replica (s_2), the storage cost of a new snapshot dominates the IO cost of replay for EC2. Therefore all replicas are always spawned from the original s_0 snapshot in the public cloud. Instances are also not stopped between the two peaks as instances are paid for a full hour, pausing and restarting them 10 minutes later costs more than letting them run.

Table 14. Provisioning algorithm performance for private and public clouds in terms of cost and missing replica minute (MRM).

Provisioning algorithm	Private Cloud		Public Cloud (EC2)	
	Cost (time)	MRM	Cost ($)	MRM
Reactive15m	381m42s	17.5	18.29	27.2
Reactive1h	360m30s	25.8	5.00	33.7
Reactive2h	410m	42.1	4.61	41.5
Overpro6	720m	0	8.39	0
Dolly10m	381m54s	0	7.16	0
Dolly30m	352m	0	3.73	0
Dolly2h	352m	0	3.73	0

In summary, we have shown that Dolly with a prediction window as short as 30 minutes is able to provide optimal resource utilization (according to administrator defined cost functions) while always providing the required capacity.

7. Related Work

Much of the prior work on dynamic provisioning [28], [29], [30], [6] has focused on dynamic provisioning of the front tiers of web applications. In this work we focus on the database tier that differs from other tiers due to its large dynamic state. Commercial solutions such as Oracle RAC [18] use a shared disk approach to avoid the state replication problem. The use of in-memory databases on top of a shared storage has also been considered [18]. Our work focuses on cloud environments where a shared disk approach cannot typically be deployed.

Amazon Relational Database Service (RDS) [2] works with Amazon Auto Scaling [1] to provide reactive provisioning of asynchronously replicated (i.e. master/slave) MySQL databases based on static thresholds. Microsoft in its Azure PaaS (Platform as a Service) cloud offering provides built-in replication in the lower layer of its platform but hides it to the user [22]. Provisioning could be enhanced on both platform using Dolly.

The few papers related to dynamic provisioning of databases usually focus on workload prediction without modeling the time to spawn new replicas [11]. Dolly can work with any load predictor and provisions database replicas accordingly by predicting VM cloning and replica resynchronization time. The problem of re-synchronizing database replicas in a shared nothing environment has been described in [25]. However, the proposed technique only relies on log replay and does not exploit snapshotting as a way to bring up new replicas. Even in a more recent work [14], state synchronization time is based on fixed estimates for replay. We have shown that using virtualization, we are able to snapshot databases via VM cloning and predict state replication time accurately.

8. Conclusion

Database provisioning is a challenging problem due to the need to replicate and synchronize disk state. Since modern data centers and cloud platforms employ a virtualized architecture, we proposed a new database replica spawning technique that leverages virtual machine cloning. We argued that VM cloning offers a replication time that depends solely on the VM disk size and is independent of the database size, schema complexity and database engine. We proposed models to accurately estimate replica spawning time and analyzed the tradeoffs between capacity provisioning and database state snapshotting. To the best of our knowledge, Dolly is the first database provisioning system that can be adapted to the specifics of various cloud platforms via administrator-defined cost functions.

We implemented Dolly and integrated it with a commercial-grade open source database clustering middleware. We proposed different cost functions to optimize resource usage in a private cloud and to minimize cost for the Amazon EC2 public cloud. We evaluated our prototype with a TPC-W e-commerce workload and demonstrated the benefits of an automated database provisioning system for the cloud, with optimized solutions adapted to different cloud platform specifics. We plan to release Dolly as open source software and hope that it will facilitate replicated database deployments in virtualized environments such as clouds.

Acknowledgement

We would like to thank Steve Dropsho for early contributions to this work. This research was supported in part by NSF grants CNS-0834243, CNS-0720616, CNS-0916972, CNS-0855128, and a gift from NEC.

9. REFERENCES

[1] Amazon Auto Scaling - http://aws.amazon.com/autoscaling/

[2] Amazon RDS - http://aws.amazon.com/rds/

[3] C. Amza, E. Cecchet, Anupam Chanda, Alan L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel – *Specification and implementation of dynamic Web site benchmarks* – WWC, 2002.

[4] C. Amza, A. Cox and W. Zwaenepoel – *Conflict-Aware Scheduling for Dynamic Content Applications* – USITS'03, Seattle, WA, March 2003.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, I., and A. Warfield – *Xen and the art of virtualization* – SOSP, October 2003.

[6] M. N. Bennani and D. A. Menasce – *Resource allocation for autonomic data centers using analytic performance models* – ICAC '05, Washington, DC, USA, 2005.

[7] J. Blancet – *Snapshots in Xen* – Online FAQ, https://zagnut.storeitoffsite.com/home/jim.blancet/FAQ/Snapshots%20in%20xen

[8] E. Cecchet, J. Marguerite, W. Zwaenepoel – *C-JDBC: Flexible Database Clustering Middleware* – Usenix Annual Technical Conference, Boston, MA, USA, June 2004.

[9] E. Cecchet, G. Candea and A. Ailamaki – *Middleware-based Database Replication: The Gaps between Theory and Practice.* – ACM SIGMOD, June 10-12, 2008

[10] A. Chandra, W. Gong, and P. Shenoy – *Dynamic Resource Allocation for Shared Data Centers Using Online Measurements* – IWQoS 2003, June 2003.

[11] J. Chen, G.Soundararajan, C.Amza – *Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers* – ICAC '06, June 2006.

[12] S. Elnikety, S. Dropsho and W. Zwaenepoel – *Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases* – EuroSys, March 2007.

[13] S. Elnikety, S. Dropsho, E. Cecchet and W. Zwaenepoel – *Predicting Replicated Database Scalability from Standalone Database Profiling* – EuroSys, April 2009.

[14] S. Ghanbari, G. Soundararajan, J. Chen, and C. Amza – *Adaptive Learning of Metric Correlations for Temperature-Aware Database Provisioning* – ICAC, June 2007.

[15] J. N. Gray, P. Helland, P. O'Neil, D. Shasha – *The Dangers of Replication and a Solution* – ACM SIGMOD, 1996.

[16] J. Hellerstein, F. Zhang, and P. Shahabuddin – *An Approach to Predictive Detection for Service Management* – Proceedings of the 12th IEEE International Conference on Systems and Network Management, 1999.

[17] B. Kemme, G. Alonso – *Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication* – VLDB'00, September 2000.

[18] K. Manassiev and C. Amza – *Scaling and Continuous Availability in Database Server Clusters through Multiversion Replication* – DSN 2007, June 2007.

[19] ObjectWeb RUBiS Virtual Appliance - http://rubis.ow2.org/

[20] Oracle – Oracle Real Application Clusters 11g – Oracle Technical White Paper, April 2007.

[21] OpenNebula project. http://opennebula.org/

[22] M. Otey – *SQL Server vs. SQL Azure: Where SQL Azure is Limited* - SQL Server Magazine, August 2010.

[23] C. Plattner, G. Alonso – *Ganymed: Scalable Replication for Transactional Web Applications* – ACM/IFIP/USENIX Middleware, Toronto, Canada, October 2004.

[24] Sequoia Project. http://sourceforge.net/projects/sequoiadb/

[25] G. Soundararajan and C. Amza – *Online data migration for autonomic provisioning of databases in dynamic content web servers* – 2005 Conference of the Centre For Advanced Studies on Collaborative Research, Toronto, October 2005.

[26] TPC-W Benchmark, ObjectWeb implementation, http://jmob.objectweb.org/tpcw.html.

[27] Transaction Processing Council. http://www.tpc.org/.

[28] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal – *Dynamic Provisioning for Multi-tier Internet Applications* – ICAC-05, Seattle, June 2005.

[29] D. Villela, P. Pradhan, and D. Rubenstein – *Provisioning Servers in the Application Tier for E-commerce Systems* – IWQOS 2004, June 2004.

[30] Q. Zhang, L. Cherkasova, and E. Smirni – *A regression based analytic model for dynamic resource provisioning of multi-tier applications* – ICAC '07, Washington, DC, 2007.