GINJA: One-dollar Cloud-based Disaster Recovery for Databases

Joel Alcântara, Tiago Oliveira, Alysson Bessani
LaSIGE – Faculdade de Ciências, Universidade de Lisboa
Portugal

Abstract

Disaster Recovery (DR) is a crucial feature to ensure availability and data protection in modern information systems. A common DR approach requires the replication of services in a set of virtual machines running in the cloud as backups. This leads to considerable monetary costs and managing efforts to keep such cloud VMs. We present GINJA, a DR solution for transactional database management systems (DBMS) that uses only cloud storage services such as Amazon S3. GINJA works at file-system level to efficiently capture and replicate data updates to a remote cloud storage service, achieving three important goals: (1) reduces the costs for maintaining a cloud-based DR to less than one dollar per month for relevant databases' sizes and workloads (up to 222× less than the traditional approach of having a DBMS replica in a cloud VM); (2) allows a precise control of the operational costs, durability and performance trade-offs; and (3) introduces a small performance overhead to the DBMS (e.g., less than 5% overhead for the TPC-C workload with ≈ 10 seconds of data loss in case of disasters).

CCS Concepts • **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Information systems → Data replication tools**;

Keywords Disaster recovery, Databases, Cloud

ACM Reference format:
Joel Alcântara, Tiago Oliveira, Alysson Bessani. 2017. GINJA: One-dollar Cloud-based Disaster Recovery for Databases. In *Proceedings of Middleware '17, Las Vegas, NV, USA, December 11–15, 2017,* 13 pages.
DOI: 10.1145/3135974.3135985

1 Introduction

The occurrence of disasters introduces some serious challenges to the design of IT systems. In opposition to other sources of failures, disasters affect the whole (or at least a big part of the) infrastructure where the system is hosted, resulting in greater damage to the service provided. Consequently, the ability to tolerate disasters requires specific data protection mechanisms and careful planning [32].

More specifically, tolerating disasters requires placing backup resources in a geographically separated location so that the same disaster does not affect the primary and the backup infrastructures. Such approach results in significant additional costs, and thus it is not used by budget-constrained services.

The emergence of cloud computing made it possible to implement disaster recovery (DR) with a small fraction of the costs of a dedicated infrastructure [49]. System operators can thus rely on cloud providers to host a portion (or even full copies) of their system and, if the primary site goes offline, they can quickly assume the service provision.

Cloud-based disaster recovery mechanisms require different approaches to deal with stateless and stateful services. For the former, administrators only have to store server VM images to enable the services to be started when required. For *stateful services such as databases*, there are basically two options: periodically storing state snapshots, or maintaining a warm backup on the cloud [34]. The first approach is known as *Backup and Restore* while the later is sometimes called *Pilot Light*, in the sense that this backup replica can spark a whole backup infrastructure if needed [41]. The replication protocol for maintaining such replica in the cloud can be implemented at different layers, such as within the service itself [11, 14, 33], in the virtualization platform [40, 50], or at the storage level [31, 39].

Despite all these options, data loss is still a common event with severe consequences. Although statistics about data losses and its effects are sometimes misleading [27], recent surveys showed that data loss costs $1.7 Trillion per year for medium and big companies [35]. Few years ago a survey by Symantec showed that 40% of Small and Medium Enterprises (SMEs) do not do regular backups [44]. We believe the situation improved in the last years, but it is unlikely that this protection gap disappeared. A more recent survey revealed that 58% of the SMEs could not sustain any amount of data loss [29], and that 62% of these companies do not backup their data on a daily basis. These numbers clearly indicate that even simple backup routines are still a challenge for SMEs, and reveal that fully automated disaster recovery solutions are not yet widely deployed. This landscape is even worse if one considers new data loss threats, such as ransomware [18, 45]. *Lack of budget and automation* are usually pointed as key challenges for implementing effective business continuity plans [1].

In this paper, we improve this situation, specially for SMEs and other organizations that cannot afford the cost and complexity of geo-replication and existing DR solutions, through the exploitation of two facts. First, non-VM-based (also called "serverless") cloud services have the potential of reducing the costs and management complexity[1] of a disaster recovery solution. Second, most businesses critical data is stored in database management systems (DBMS), therefore, it is paramount for any serious DR solution to protect these systems.

We present GINJA, a disaster recovery system for transactional DBMS based on popular cloud object storage services such as Amazon S3, Azure Blob Storage or Google Storage. GINJA was designed with five objectives in mind: low operational costs, fine-grained control over the data that can be lost due to a disaster, low performance overhead, ease of use, and portability among different DBMS.

[1]It's worth to recall that having a database replica on a cloud VM requires, besides the DBMS configuration, a proper setup of a public IP, firewalls, security updates, etc.

To the best of our knowledge, GINJA *is the first system to exploit a new region in the design space of disaster tolerance/recovery systems*, right between *Backup and Restore* and *Pilot Light* solutions. In particular, it has costs close to the former (i.e., maintaining backup database snapshots in the cloud, without requiring running a VM) with the same control over data loss and performance of the later (i.e., having a warm database replica in a cloud VM). To do that in a portable way, we had to overcome several challenges: (1) define means to capture all the relevant I/O from the DBMS, (2) map these updates to a data model that fits the clouds object storage interface, and (3) provide algorithms for controlling the behavior of the system to match different requirements and budgets.

Although there is a large body of work on database replication for fault tolerance [33], these works are mostly orthogonal to GINJA as they consider the *coordination of database replicas*. In contrast, our challenges are more related with *the practical aspects of capturing database disk updates and writing them in a passive remote storage in a cost-efficient way*, keeping a tight control over the cost vs. performance vs. maximum data loss trade-off.

Besides the obvious benefits that GINJA brings to small-to-medium DBMS users, it can also be employed for improving the availability of cloud database services. Currently, services like Amazon Relational Database Service offers Multi-AZ (Availability Zone) instances that synchronously replicate database updates to another availability zone within the same region [3]. However, *the cost of this setup is roughly twice the cost of running a single-AZ instance and it does not support more severe failure scenarios* that would require replication across regions or providers. GINJA supports provider-scale disaster tolerance [19] by exploiting the object storage services available in most cloud providers, protecting thus cloud databases from cloud outages [28].

We have implemented and evaluated a prototype of GINJA supporting PostgreSQL [13] and MySQL [10] to show that our solution is feasible, cost-efficient, and presents acceptable performance overheads. For instance, we are able to provide DR for many database setups relevant to SMEs (e.g., databases with up to 40GB of size and tens of updates/minute) costing only *one dollar/month*, which is $48\times$ less than the cost of running the cheapest EC2 VM indicated for small to mid-size databases (m3.medium [2]) for a month. Furthermore, our results show that the use of GINJA leads to a small DBMS performance loss when running the TPC-C benchmark, and minor additional CPU and memory usage on the database server. Although our current implementation only supports PostgreSQL and MySQL, it can be extended to support other DBMS.

In summary, this paper makes the following contributions:

1. It shows that cloud storage services like Amazon S3 are not only cost-effective infrastructures for keeping remote backups, but also solutions for low-cost tightly-controlled DBMS disaster recovery (§3);
2. A set of algorithms for replicating database transactions and checkpoints to a remote cloud storage with fine-grained control of the cost vs. performance vs. data loss trade-off (§5);
3. A detailed monetary cost model for implementing cloud-backed disaster recovery of transactional databases (§7);
4. An implementation supporting the two most popular open-source relational databases (§4 and §6) and its evaluation using a real cloud provider (§8).

2 Disaster Recovery

A *Disaster* is an event that has a negative impact on organizations business continuity and/or finances [41]. Examples of disasters include network and power outages, hurricanes, earthquakes, floods, and so forth. *Disaster Recovery* is the area that makes IT systems tolerant and recoverable from the damages caused by disasters. This is mainly achieved by having a *Primary Site* infrastructure to respond in normal operation, plus a *Secondary Site* (or backup site) in a geographically-distant location [24, 32].

Yet, different systems have different disaster recovery requirements [39]. Such requirements include recovery time, consistency degree of the data recovered, performance impact during normal operation, distance between primary and secondary sites, and costs. In practice such requirements define two parameters [24]: *Recovery Point Objective* (RPO), which is the amount of updates (measured in time) that can be lost due to a disaster; and *Recovery Time Objective* (RTO), which refers to the duration of downtime that is acceptable before a system recovers from a disaster.

There are several ways to implement a disaster recovery strategy [32]. The classical approach is the *Backup and Restore* [42]. This technique consists of periodically taking consistent snapshots of the data (optionally interspersed with incremental backups), and writing them in storage devices kept off site. Although this approach is attractive for being low-cost, it has the disadvantages of having long recovery time and always restoring the system to an outdated state, as backup intervals are typically long. An alternative strategy is *Remote Mirroring* [31]. In this approach, the system continuously replicates its data to an online remote mirror (also called *Pilot Light*), which ensures the continuity of the system if a disaster occurs. Despite being usually more expensive, this technique can substantially reduce both the RPO and RTO when compared with the *Backup and Restore*.

The data replication between sites can be performed essentially in two ways: synchronously or asynchronously [22, 50] (also called eager and lazy replication in the database community [33]). In *Synchronous Replication*, the system loses performance as the primary site can only return successfully from a write operation after it has been acknowledged by the secondary site. In *Asynchronous Replication* the primary site is allowed to proceed its execution without waiting for the synchronization between sites to complete. This type of replication overcomes the performance limitations of synchronous replication at the expense of allowing recent updates to be lost if a failure occurs.

Public clouds are an appealing solution for implementing DR mechanisms. The main reasons are their large portfolio of services (e.g., object storage, computing, networking, database, queue services), relatively user-friendliness, security, multi-site availability, and the pay-as-you-go cost model. These factors allow the design of DR solutions suitable for each organization regarding its objective (RPO and RTO) and budget [41]. The simplest (and probably the cheapest) example is the storage of data backups in cloud storage services such as Amazon S3. A more evolved (and expensive) solution considers a subset of services replicated in VMs running in the cloud (e.g., on Amazon EC2), providing lower data loss and recovery time in case of a disaster.

Some public clouds also provide disaster recovery services that typically use their infrastructures as a secondary site. Examples of such services are Azure Site Recovery [8] and vCloud Air Disaster Recovery [16]. These services automate the configuration and management of the cloud backup infrastructure, but their cost is equivalent or even higher than maintaining plain backup VMs in the cloud. It is also possible to run the primary site of a system entirely in a cloud. However, this approach does not eliminate the need for disaster recovery since cloud-wide outages, although rare, are a potential threat to systems that rely entirely on one cloud infrastructure to perform its functions [28].

The more cloud resources a system requires in failure-free operation, the higher will be the costs of the DR solution. Even if the costs during failover are slightly higher in cloud-based solutions, the overall costs can still be smaller since disasters are supposed to be rare events [49].

3 Low-cost Cloud-based Disaster Recovery

In this paper, we advocate that current cloud-based disaster recovery solutions are much more expensive and difficult to manage than they should be. In this sense, we believe that a full-fledge database disaster recovery system could be implemented without requiring any dedicated VM in the cloud. Such a system can work as follows. Initially, a copy of all database files is uploaded to a cloud storage service (e.g., Amazon S3), and then, as updates are committed to the log file, the system sends them to the cloud as commit objects. As new updates keep being performed, the DBMS executes a checkpoint to update the table files and to clean its commit log. In this situation, the system updates the database files in the cloud and removes outdated commit objects. In case of disaster, the recovered database needs to be able to figure out the pre-disaster state using the objects stored in the cloud.

Such a system can be extremely cheap if one accepts to lose a few recent updates in case of a disaster (as in most DR solutions). This would enable the DR system to send batches of updates to the cloud periodically.

As an illustrative example, consider that someone wants to spend a maximum of *\$1 per month* in a database DR solution. In May 2017, Amazon S3 standard storage costs are \$0.023 per GB/month, \$0.005 per 1000 file uploads, and free upload bandwidth and delete operations [4].[2] Considering this, it is possible to plot the capacity of a database (in terms of size and number of cloud synchronizations per hour) for such one-dollar budget, as shown in Figure 1.

In the figure, every point below the line represents a setup costing less than \$1 per month. For example, this budget is enough to protect a database with 4.3GB with four synchronizations per minute (setup C), or a 20GB database with two synchronizations per minute (setup B), or even a 35GB database synchronized once every 72 seconds (setup A). Importantly, an organization whose activity happens mostly from 9AM to 5PM (which is the case for many non-online business) can have roughly three times more synchronizations per hour during this period.

Notice that these setups still provide acceptable RPOs for many medium-size organizations. In any case, by understanding what one can have with \$1 per month, it is possible to assess the cost of more demanding setups (i.e., larger databases or smaller RPOs).

[2]Other services such as Azure Storage, Google Storage, and Rackspace Files offer similar price models. GINJA can be used with any of them.

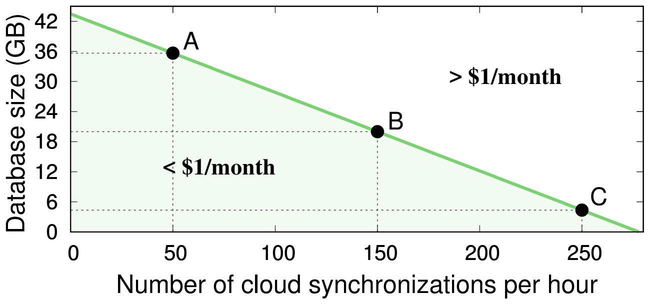


Figure 1. Database size and number of cloud synchronizations per hour in an S3-based DR solution with a \$1 monthly budget.

The system presented in this paper, GINJA, exploits this opportunity to enable any small and medium organization running a relational DBMS to have a DR solution almost for free, and with close-to-zero management effort.

4 Transactional Database I/O

GINJA is a DR solution for database management systems. The integration between our system and the DBMS happens at the file system level. This allows us to intercept every file system call performed by the DBMS on the database-related files. In this section, we describe the kind of DBMS our system assumes and discuss how PostgreSQL and MySQL fit in this model.

We consider transactional databases that implement data durability using a set of *table files* and a Write-Ahead Log (WAL) divided in several *segment files* [25, 37]. The I/O on these files is performed on the granularity of a *page*, which is composed by many *records*, each one storing a database update. Every time a transaction is committed, the only important I/O performed is a synchronous write to a WAL file segment. All the table pages remain in memory until a periodic checkpoint occurs. When this happens, the pages are written to the table files, and a special record is inserted in the WAL marking that everything before this record is already in durable memory.

Implementing a DR solution with fine-grained control of the database RPO without changing the DBMS requires a deep understanding on how databases access these files. More precisely, there are at least three types of events that need to be detected. The first one is an update commit, when a record is written to the WAL. The second one corresponds to the write that marks the beginning of a checkpoint. The last type of event we need to detect is the last write of a checkpoint, i.e., the last write after which it is safe to delete old WAL entries. Table 1 describes these events for the databases we use in this paper: PostgreSQL and MySQL.

PostgreSQL [14, 43] keeps its log segments in a set of `x_log` files (with pages of 8kB), and periodically (with a configurable period), writes the dirty pages (also 8kB) to the table files. Additionally, it uses a `pg_log` file to store the status of each transaction (the checkpoint starts with a write in this file) and a small `pg_control` file to store a pointer to the last checkpoint record in the WAL, marking the starting point on the WAL upon a recovery. A write to `pg_control` marks the end of a checkpoint.

MySQL supports different types of storage engines. In this work we consider only InnoDB, the standard engine for supporting ACID semantics [10]. MySQL/InnoDB (or simply MySQL) writes all committed transactions to an `ib_logfile` file (in pages of 512 bytes),

Table 1. How GINJA detects the three most important DBMS events in PostgreSQL and MySQL. * Except the header of the `ib_logfile0`.

Event	PostgreSQL	MySQL
Update commit	sync. write to a `pg_xlog` file	sync. write in one of the `ib_logfile` files*
Checkpoint begin	sync. write to a `pg_clog` file	sync. write to one of the data files (`ibdata`, `.ibd`, and `.frm`)
Checkpoint end	sync. write to the `global/pg_control` file	sync. write in the offset 512 and/or 1536 of the `ib_logfile0` file

and executes checkpoints quite differently from PostgreSQL. More specifically, the system can flush modified database pages (of 16kB) to their respective files at any moment, in small batches. This mechanism is known as fuzzy checkpoint [9]. The fact checkpoints are "opportunistic" makes their write pattern a bit more complicated and variable than the ones in PostgreSQL. However, as can be seen in Table 1, it is possible to detect the begining and end of these checkpoints by verifying a handful of conditions.

It is important to highlight that even with the most write-intensive workloads, the capture of these events always allow the recovery of the database to its "committed" state right before the occurrence of a crash. This happens because, by capturing these events before a crash, it is possible to reconstruct the table and segment files in such a way that the DBMS can rebuild its state using its crash-recovery capabilities. For example, the DBMS can read (in the `pg_control` file for PostgreSQL and in the offset 512 or 1536 of the `ib_logfile0` file for MySQL) where the last checkpoint is, and then apply all the WAL records after that.

5 Ginja

GINJA can be seen as a transparent middleware that intercepts the I/O performed by the DBMS and backs up the relevant data to a cloud storage service in a cost-efficient manner. Although our implementation consists in an application-specific FUSE file system (see §6 for details) able to capture the semantics of the database's I/O operations without having to change the DBMS, our design is generic and only assumes that the events of Table 1 are intercepted. Therefore, nothing prevents it from being integrated on a database or even implemented on the kernel of an operating system.

GINJA relies on cloud storage services (e.g., Amazon S3, Azure Blob Storage) to store its data in a remote site. As described before, we choose such services as secondary infrastructure because they have the potential of lowering both the monetary and management costs of our DR solution.

This design puts GINJA apart from existing works on cloud disaster recovery [31, 36, 39, 50], and impacts our system in three important ways. First, storage clouds provide REST interfaces containing only a few basic operations (PUT, GET, LIST, and DELETE). Consequently, we have to implement all DR control at the primary side (i.e., at the client side). Second, we must make as few assumptions as possible about the underlying storage clouds, so that our clients can choose the cloud provider they want with few or no modifications to our code. Finally, it is crucial that we take into account the pricing model of the cloud storage services when performing cloud operations, to reduce costs as much as possible.

It is worth to remark that GINJA is not a complete disaster recovery solution. For instance, our system does not consider the detection of a failure on the primary infrastructure and the switching to a backup. Although there are works that address this problem (e.g., [40]), the deployment of a fully-automated disaster recovery system is highly dependent on the services being protected and the procedures defined in the organization disaster recovery plan.

5.1 Controlling Costs and Data Losses

GINJA deals with the fundamental trade-off between performance and data protection by allowing users to decide the maximum amount of recent updates that can be lost when a disaster occurs. Thus, instead of following a completely synchronous or asynchronous approach, we define a model that allows users to choose the desired synchronization level. Furthermore, as sending data to the cloud has its costs, our model also delegates to users the performance-cost trade-off. This model includes two parameters:

- *Batch* – the maximum number of database updates included in each cloud synchronization;
- *Safety* – the maximum number of database updates that can be lost in the event of a disaster.

These parameters define a threshold of database updates that trigger GINJA to perform its actions. More precisely, *Batch* defines how often WAL writes are sent to the cloud, whereas *Safety* defines the durability guarantees provided by GINJA.

Batch and *Safety* can be defined both in terms of number of updates – B and S – and time – T_B and T_S – working as follows. A *batch of updates is sent to the cloud* if B updates are executed or if there are some updates to be sent to the cloud and T_B seconds have elapsed since the last synchronization ended. Similarly, a *WAL write performed by the database blocks* if there are more than S updates that are not confirmed to have been written to the cloud or if there are some updates to be sent to the cloud and T_S seconds have elapsed since the first non-synchronized update was executed. Notice that in write-intensive workloads, only B and S will be relevant since timeouts will not be triggered.

Figure 2 illustrates how these parameters work. Whenever B operations are executed in the DMBS, GINJA performs a cloud synchronization and allows the database management system to proceed its normal operation. On the other hand, when the S^{th} database update since the last successful synchronization is submitted (U_{21} in the figure), our system blocks the DBMS until the pending cloud synchronizations are confirmed.

Ideally, B should be substantially lower than S so that GINJA does not block or interfere with the DBMS performance during regular operation.

5.2 Data Model

GINJA data model allows the synchronization of file updates as they are issued locally, and the reconstruction of those files from the objects present in the cloud when necessary. This model aims to reduce the total volume of data kept in the cloud, and to minimize

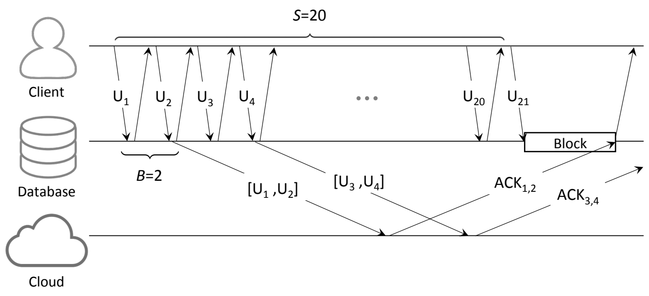


Figure 2. Influence of B (*Batch*) and S (*Safety*) in the execution of GINJA. In this example $B = 2$, thus each cloud backup includes two database updates. GINJA blocks the DBMS whenever more than $S = 20$ database updates are executed without being acknowledged by the cloud.

the number of cloud operations executed. The data model considers two types of objects:[3]

- *WAL Objects* contain data written to the local WAL segments. The content of each local WAL segment is stored in several WAL objects (one for up to B updates). The WAL objects are named following the format `WAL/<ts>_<filename>_<offset>`, in which the `ts` establishes total order on the WAL objects, `filename` is the name of the corresponding WAL segment, and `offset` is the position of its content in the segment.
- *DB Objects* store information relative to all relevant database files excluding the WAL segments. There are two types of DB objects: *dumps* and *incremental checkpoints*. The DB objects are named following the format `DB/<ts>_<type>_<size>`, containing thus its `ts`, `type` (*"dump"* or *"checkpoint"*), and `size`. In this case, the `ts` corresponds to the timestamp of the last uploaded WAL object before the beginning of the checkpoint.

5.3 Algorithms

This section details GINJA algorithms for initialization and recovery, update processing, and checkpoint management.

Initialization. Algorithm 1 describes how GINJA is initialized in its different modes (*Boot*, *Reboot* and *Recovery*). When started, and before engaging in one of its three initialization modes, the system initializes an empty *cloudView* data structure (used to keep track of the WAL and DB objects in the cloud) and starts all the required threads in the system (Lines 2–6).

The *Boot* mode is used to create a dump of an existing database on the cloud. Concretely, the system creates a set of WAL objects (one for each local WAL segment), and one dump DB object (Lines 7–18). Only after all the objects are successfully uploaded to the cloud the file system is mounted and the DBMS can be started.

The *Reboot* mode should be used to restart the system after a safe stop of the DBMS. This mode assumes that the data on the cloud is synchronized with the local files of the database. Therefore, the only required step is to update the *cloudView* by listing the objects

Algorithm 1: Initialization tasks.

```
1  cloudView ← ∅;                                          // Used in all Algs.
2  TaskT_B.startTimer(T_B);                                 // Used in Alg. 2
3  TaskT_S.startTimer(T_S);                                 // Used in Alg. 2
4  for 1 ≤ i ≤ nThreads do
5      CommitThread_i.start();                              // Used in Alg. 2
6  CheckpointThread.start();                                // Used in Alg. 3
7  Mode Boot begin
8      currentTs ← 0;
9      for each file in Local WAL Segments, in increasing order do
10         objName ←"WAL/"+currentTs+"_"+file.name+"_0";
11         cloud.PUT(objName,file.content);
12         cloudView.addWAL(currentTs,file.name,0);
13         currentTs ← currentTs + 1;
14     dbObject ← ∅;
15     for each file in Local DB Files do
16         dbObject.add(file.name,file.content);
17     cloud.PUT("DB/0_dump_"+dbObject.size,dbObject);
18     cloudView.addDB(0,"dump",dbObject.size);
19 Mode Reboot begin
20     cloudList ← cloud.LIST();
21     for each obj in cloudList do
22         cloudView.add(obj);
23 Mode Recovery begin
24     cloudList ← cloud.LIST();
25     for each obj in cloudList do
26         cloudView.add(obj);
27     dump ← cloud.GET(mostRecentDump(cloudList.dbObjects));
28     for each file in dump do
29         writeLocally(file.name,0,file.content);
30     checkpoints ← newerThan(cloudList.dbObjects,dump.ts);
31     maxCkptTs ← dump.ts;
32     for each obj in checkpoints, in increasing ts order do
33         currentCkpt ← cloud.GET(obj);
34         for each file in currentCkpt do
35             writeLocally(file.name,file.offset,file.content);
36         maxCkptTs ← obj.ts;
37     segments ← newerThan(cloudList.walObjects,maxCkptTs);
38     for each obj in segments, sortedby ts and with no gaps do
39         content ← cloud.GET(obj);
40         writeLocally(obj.filename,obj.offset,obj.content);
```

present in the cloud (Lines 19–22). Notice that even this step might be removed if this data structure is persisted during a safe stop.

The *Recovery* mode is used to rebuild the database files from the objects stored in the cloud. The first step is to list the objects in the cloud and update the *cloudView* data structure (Lines 24–26). Then, the database files are reconstructed from the most recent dump in the cloud (Lines 27–29) and, afterwards, these files are updated with the incremental checkpoint objects (Lines 30–36). Finally, GINJA downloads the WAL data objects written after the last checkpoint and rebuild the local WAL segments following the *ts* ordering so that the DBMS can be restarted (Lines 37–40).

[3]We limit the maximum size of each cloud object to a configurable limit (20MB by default) to optimize the upload latency [30]. This feature is not shown in the algorithms for the sake of simplicity.

Algorithm 2: Database Commits.

```
1  $commitQueue \leftarrow \emptyset$;    // Holds all the pending synchronizations
2  $timeoutT_S \leftarrow false$;
3  $timeoutT_B \leftarrow false$;
4  When write(WAL_segment, offset, content) is intercepted begin
5      $writeLocally(WAL\_segment, offset, content)$;
6      $commitQueue.put(\langle WAL\_segment, offset, content \rangle)$;
7      wait until $commitQueue.size \leq S$ and $timeoutT_S = false$;
8  CommitThread Execution begin
9      Loop
10         wait until $commitQueue.size \geq B$ or $timeoutT_B = true$;
11         $updates \leftarrow commitQueue.getNextBatch()$;
12         $aggUpdates \leftarrow aggregateUpdates(updates)$;
13         for each $u$ in $aggUpdates$ do
14             $ts \leftarrow cloudView.getNextWALts()$;
15             $cloud.PUT($"WAL/"$+ts+$"_"$+u.filename+$"_"$+u.offset)$;
16             $cloudView.addWAL(ts, u.filename, u.offset)$;
17         $TaskT_B.resetTimer()$;
18         $timeoutT_B \leftarrow false$;
19         wait until $commitQueue.lastBatchElements() = updates$;
20         $commitQueue.removeLastNElements(updates.size)$;
21         $TaskT_S.resetTimer()$;
22         $timeoutT_S \leftarrow false$;
23 TaskT_B (upon timeout) begin
24     if $commitQueue.size > 0$ then
25         $timeoutT_B \leftarrow true$; // Trigger an upload
26 TaskT_S (upon timeout) begin
27     if $commitQueue.size > 0$ then
28         $timeoutT_S \leftarrow true$; // Block the DBMS
```

Algorithm 3: Checkpoints and Garbage Collection.

```
1  $checkpointQueue \leftarrow \emptyset$;
2  $timestamp \leftarrow \emptyset$;
3  When write(dbFile, offset, content) is intercepted begin
4      if $\langle dbFile, offset, content \rangle$ is the first write in checkpoint then
5          $timestamp \leftarrow cloudView.getLastWALts()$;
6      $writeLocally(dbFile, offset, content)$;
7      $dbObject \leftarrow addAndAggregate(\langle dbFile, offset, content \rangle)$;
8      if $\langle dbFile, offset, content \rangle$ is the last write in checkpoint then
9          if $cloudView.getTotalDBSize() \geq 150\% \times$ local DB size then
10             $dbObject \leftarrow$ create dump from local DB files;
11             $dbObject.type \leftarrow$ "dump";
12         else
13             $dbObject.type \leftarrow$ "checkpoint";
14         $dbObject.ts \leftarrow timestamp$;
15         $checkpointQueue.add(dbObject)$;
16         $dbObject \leftarrow \emptyset$;
17 CheckpointThread Execution begin
18     Loop
19         wait until $checkpointQueue.size > 0$;
20         $obj \leftarrow checkpointQueue.remove()$;
21         $cloud.PUT($"DB/"$+obj.ts+$"_"$+obj.type+$"_"$+obj.size, obj)$;
22         $cloudView.addDB(obj.ts, obj.type, obj.size)$;
23         for each $walObject \in cloudView : walObject.ts \leq obj.ts$ do
24             $cloud.DELETE(walObject.objectName)$;
25             $cloudView.delete(walObject)$;
26         if $obj.type =$ "dump" then
27             for each $dbObject \in cloudView : dbObject.ts < obj.ts$ do
28                 $cloud.DELETE(dbObject.objectName)$;
29                 $cloudView.delete(dbObject)$;
```

Database Update Commits. Algorithm 2 describes how GINJA processes the intercepted writes to WAL segment files without violating the parameters B, S, T_B, and T_S.

When the system intercepts an update to a WAL segment file, it writes the data on the local copy of the file and enqueues the update to be sent to the cloud (Lines 4–6). The operation only returns to the DBMS if the S and T_S parameters are not violated, otherwise the systems blocks (Line 7) until the pending writes are uploaded.

Lines 8–22 show how commits are processed. First, the writes are aggregated respecting B and T_B (Lines 9–12), resulting in one or more WAL objects, depending on the number of segments affected by the committed writes.[4] After being aggregated, they are sent to the cloud (Lines 13–16).

The aggregation is important because the DBMS write to the log on the granularity of a page, and many times these pages are overwritten with more updates. Consequently, by aggregating them we coalesce many updates in a single cloud object upload. This reduces the storage used and the total number of PUT operations executed in the cloud, resulting in a significant decrease in the monetary cost of our DR solution.

GINJA uses multiple *CommitThreads* to upload objects to the cloud in parallel (see Lines 4–5 of Algorithm 1), achieving great benefits in terms of performance [30]. However, this parallelism makes it no longer guaranteed that WAL objects are uploaded following their timestamp order (i.e., the *ts* obtained on Line 14). In the worst case scenario, a disaster may occur in the moment when the most recent WAL updates are already replicated in the cloud, while others with smaller timestamps are still in transmission. During *Recovery*, GINJA deals with this incomplete state by downloading only the WAL objects that have consecutive timestamps. Consequently, to limit the maximum number of updates lost in case of failure to S, GINJA blocks the DBMS until all WAL objects with consecutive *ts* values are uploaded. This can be observed in Algorithm 2: the variables that control these parameters (specifically `commitQueue.size`, `timeoutT`$_S$, and the timer of `TaskT`$_S$) are reseted (unblocking the DBMS) if all WAL objects previously uploaded can be used to recover from a disaster that would occur immediately (Lines 20–22).

Checkpoints and Garbage Collection. Algorithm 3 describes how GINJA handles checkpoints. As performance is one of our key concerns, we decouple as much as possible the (local) DBMS checkpoints from the writing of checkpoints to the cloud (Lines 6, 20–21). Therefore, checkpoint data is collected as the files are updated during a checkpoint (Lines 3–16) and, when the checkpoint is finished locally, a separate thread is used to send the updates to the cloud as DB objects (Lines 17–29). Notice the checkpoint begin and end conditions (see Table 1) are verified in Lines 4 and 8, respectively.

[4]WAL segments are typically much larger than the page size (e.g., 16MB vs. 8kB in PostgreSQL and 48MB vs. 16kB in MySQL). Consequently, this aggregation typically results in only one cloud object.

There are two ways of sending checkpoint data to the cloud: as a *checkpoint* or as a *dump*. Whenever the total size of the DB objects in the cloud is greater than or equal to 150% of the local database size, GINJA creates a new database *dump* (Lines 9–11). Otherwise, it creates an incremental *checkpoint*. In the first case, GINJA will not execute any write in the local DB files while the *dump* object is being created, to guarantee that the database is dumped in a consistent way. This does not block database commits as WAL file writes are mostly independent of checkpoint processing (at least in the two databases we support).

Every time a DB object with timestamp *ts* is completely uploaded to the cloud, GINJA removes all WAL objects with timestamps up to *ts* (Lines 4–5 and 23–25). This is safe because such WAL objects contain information that will not be used in a recovery. Additionally, when the uploaded DB object is a *dump*, all the previous DB objects (incremental checkpoints and the previous dump) are deleted as well (Lines 26–29).

5.4 Additional Features

In the following we describe some extensions to the algorithms presented in previous section.

Compression, encryption and integrity. GINJA supports the compression and/or encryption of WAL and DB objects before their write to the cloud. Compression decreases the data size and is straightforward to implement. Encryption, on the other hand, requires the management of a local secret key that cannot be stored in the cloud to preserve the database confidentiality. GINJA uses a key generated from a password (assumed to be kept secure) provided during the initialization of the system. At runtime, this key is kept in memory and never written to any local or remote file.

Our system also implements some basic integrity protection by storing a MAC of each object together with it. If encryption is enabled, the provided password is also used to generate the MAC key, otherwise, a default string (a configuration parameter) is used to generate this key.

Point-in-time recovery. The garbage collection algorithm discussed in the previous section deletes all outdated objects when a new checkpoint is written to the cloud. However, the algorithm can be modified to delete only certain objects and keep others to allow the recovery of the system to a certain point in time. More specifically, Lines 23–29 of Algorithm 3 can be modified to keep the database state on date-time T by finding the first object o stored in the cloud after T and keeping (1) the most recent dump d written before this object, (2) all incremental checkpoints written between d and o, and (3) all WAL objects written between the last incremental checkpoint and o.

As expected, storing snapshots for point-in-time recovery might substantially increase the cloud storage costs, especially for large databases. However, this is fundamental for ensuring some protection against operator mistakes and even ransomware attacks, such as the recent WannaCry virus, that ravaged many companies [18].

Backup verification. One of the key concerns in every disaster recovery plan is how to ensure the plan will actually work when a disaster strikes. An important feature of GINJA is that it allows the verification of a database backup in an easy and cheap way, without interfering with the production system.

To do that we just need to start a replica of the database in recovery mode and run a set of service-specific tests. This implies in a sequence of three validations:

1. GINJA validates the integrity of every object downloaded from the cloud through its MAC verification;
2. When restarting the system, the DBMS itself verifies the integrity of the tables and WAL segments rebuilt by GINJA;
3. Once the DBMS is started, a pre-prepared script can run a series of queries to assess if recent updates are available on the database. This verification can be made automatically using some service-specific heuristic and the result of the script can be sent to an administrator for verification.

The cost of database verification is basically the cost of downloading the database objects to a local machine or the cost of running a VM in the same cloud (if appropriate). In any case, the verification procedure can be fully automated.

6 Implementation

GINJA was implemented as a File System in User Space [46] using approximately 4000 lines of Java code distributed in 32 files. Most of this code is DBMS-agnostic and there are only two small modules that are specific for processing I/O from PostgreSQL and MySQL/InnoDB, with around 200 lines of code each. The cloud synchronization module is based on an external library capable of accessing multiple cloud storage providers [38]. Although not discussed in this paper, our system supports the replication of objects in multiple clouds, for tolerating provider-scale failures [19]. Furthermore, our current prototype implements compression using ZLIB configured for fastest operation, encryption using AES with 128-bit keys, and MACs using SHA-1.

Figure 3 presents the internal architecture of GINJA. The FS Interpreter implements a FUSE-J interface [6], and is responsible for three main tasks: (1) intercept the file system calls performed by the DBMS; (2) execute the FS operation in the local disk; and (3) forward a well-formatted data to the database processor. In this way, GINJA can be easily extended to support other DBMS by implementing new processors.

The implementation of a processor is a relatively simple and straightforward procedure. However, this requires an in-depth knowledge of the DBMS I/O management. The processor uses two different queues to put the data received from the file system: one for the WAL writes and another for the checkpoint writes.

The write operations performed in the WAL are sent to a queue named *CommitQueue*. This data structure has a maximum capacity of S elements, and only supports getting B elements at a time. Any attempt to put an element into a full *CommitQueue* will block. Likewise, attempts to take elements from a *CommitQueue* with less than B elements will result in blocking until the T_B timer expires.

The *Aggregator* thread is responsible to read sets of B updates from this queue (without removing them), aggregate those writes into a single object, and submit the resulting data to a second queue. A number of *Uploader* threads will retrieve elements from this queue and upload them in parallel as WAL objects, submitting an acknowledgment to a third queue whenever a cloud upload completes. Last but not least, a thread called *Unlocker* will remove sets of up to B elements from the head of *CommitQueue*, according to the acknowledgments received by the *Uploader* threads. In the

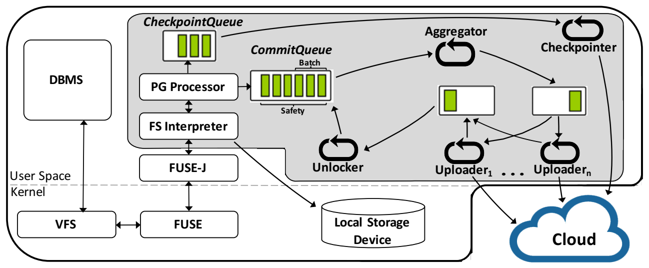


Figure 3. Architecture of GINJA.

end, the *Aggregator*, *Uploader*, and *Unlocker* threads implement Algorithm 2.

The write operations performed during checkpoints are enqueued to the *CheckpointQueue* so that a thread called *Checkpointer* aggregates the data and uploads it to the cloud in the form of DB cloud objects, implementing Algorithm 3.

7 Cost Evaluation

A key objective of our work is to reduce the operational cost of database disaster recovery. In this section, we model and evaluate the operational cost of running GINJA.

7.1 Ginja Cost Model

The factors that influence the operational cost of GINJA in the absence of disasters are the storage used to keep WAL and DB objects in the cloud, and the amount of PUT operations used to upload such objects to the cloud. Thus, the *monthly operational cost* of our system is given by the following equation:

$$C_{Total} = C_{DB\_Storage} + C_{DB\_PUT} + C_{WAL\_Storage} + C_{WAL\_PUT}$$

This equation (and the evaluation presented next) does not consider the additional storage required for point-in-time recovery. The cost of such storage can be approximated by multiplying the storage costs of the WAL and DB objects by the number of snapshots to be maintained.

Let us now explore in detail how each of the four factors of this equation can be calculated.

Storage of DB objects. GINJA uploads the information of the database files in the form of DB objects. The cost of storing these objects is given by the following equation:

$$C_{DB\_Storage} = \frac{DBSize \times 1.25}{CR} \times C_{Storage}$$

The *DB_Size* is measured in GBs and the $C_{Storage}$ in \$/GB/month. The main factor that influences this cost is the size of the database. Recall that GINJA ensures that the maximum volume that the DB objects can take in the cloud is 150% of the local database size (due to the incremental checkpoints). As a result, the average DB storage in the cloud will be 25% greater than the database size. Additionally, the DB data size can be further reduced by using compression (the compression rate, *CR* in the equation).

PUT operations of DB objects. The number of PUT operations used to upload DB objects depends essentially on how often checkpoints occur, the average checkpoint size, and the price of each PUT operation. The cost of this component can be calculated as follows:

$$C_{DB\_PUT} = \frac{30 \times 24 \times 60}{Ckpt_{Period}} \times \left\lceil \frac{CkptSize}{20MB} \right\rceil \times C_{PUT}$$

The first fraction of this equation gives us the number of checkpoints that the DBMS performs per month (note that $Ckpt_{Period}$ is given in minutes). The second fraction determines the number of PUT operations executed in each checkpoint, i.e., number of uploaded DB objects split in files of up to 20MB.

Storage of WAL objects. The third cost factor of GINJA is the volume of the WAL objects present in the cloud, calculated as follows:

$$C_{WAL\_Storage} = \left(\left\lceil \frac{W \times Ckpt_{Time}}{RecPerPage} \right\rceil + 1 \right) \times \frac{PageSize}{CR} \times C_{Storage}$$

The first part of the equation determines the maximum number of WAL segments that can be in the cloud at any moment. Recall that WAL objects written before a checkpoint are deleted from the cloud as soon as the checkpoint is completely uploaded. Consequently, the amount of storage is directly proportional to the number of updates per minute (*W* – assuming each update uses a record), and to the $Ckpt_{Time}$, which includes the checkpoint period, its duration, and the amount of time that it takes to be uploaded to the cloud.

The total number of updates performed between checkpoints is divided by the number of records per WAL page (*RecPerPage*), as we coalesce multiple writes to the same page, reaching the number of WAL segments uploaded to the cloud. The "+1" considers the worst case scenario – the situation in which the first WAL write after a checkpoint is performed in the last record of a WAL segment.

Finally, *PageSize* is the size in GB of each WAL page, *CR* is the compression rate, and $C_{Storage}$ is the storage cost.

PUT operations of WAL objects. Finally, the cost associated with the number of PUT operations of WAL objects is represented by C_{WAL_PUT}. This cost depends essentially on the database workload and the value of the parameter *B*, and is given by the following equation:

$$C_{WAL\_PUT} = \frac{W \times 60 \times 24 \times 30}{B} \times C_{PUT}$$

Every time *B* database updates are executed in the DBMS, a WAL object is uploaded to the cloud. Thus, the C_{WAL_PUT} is calculated using the number of database updates executed per month and multiplying this value by the price charged for each PUT operation.

7.2 The Cost of Running Ginja

Figure 4 presents the operational monetary costs of GINJA with different values of *B* and under different workloads. The values presented consider the usage of Amazon S3, and a database of 10GB with pages of 8kB containing 75 WAL records. We also consider that a checkpoint happens every 60 minutes and has a duration of 20 minutes, and a compression rate of 1.43 (i.e., every 1MB becomes 700kB).

The results show that the parameter *B* has a severe impact on the total monetary cost of GINJA. This can be explained by the fact that

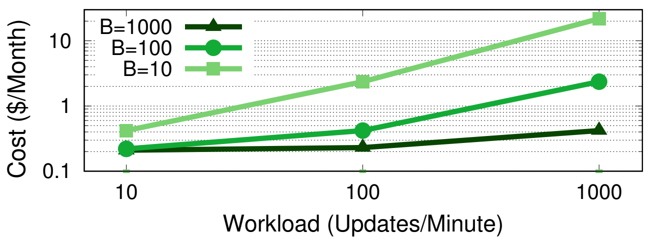


Figure 4. Effect of different configurations and workloads in Ginja's monetary cost for a 10GB database and Amazon S3.

B reduces the number of executed cloud synchronizations (i.e., PUT operations). Additionally, we can also observe that this relation is even more evident when considering more demanding update-heavy workloads.

It is worth to mention that the size of our database (10GB) implies in a fixed $C_{DB\_Storage}$ of $0.20. If one wants to consider, for instance, a 10× bigger database, this cost will be $2.

These results show that there are plenty of possible configurations that cost less than $1 per month. For reference, the cheapest VM indicated for databases in Amazon EC2 (m3.medium with Linux) costs $48.24/month in May 2017 [2].

Real application. We now present an evaluation of the costs of Ginja considering the database used in a real clinical analysis system deployed in more than 100 institutions in Europe. Table 2 presents the monetary costs of performing disaster recovery in the cloud (specifically, Amazon Web Services) using Ginja with one (RPO ≈ 1 minute) and six (RPO ≈ 10 seconds) cloud synchronizations per minute. For comparison purposes, the table also shows the cost of a DR solution based on a single backup database VM on Amazon EC2, as a *Pilot Light* for recovering the system [41]. We consider two database configurations: one hospital with a 1TB database and a workload of 630 transactions per minute, and a clinical laboratory with a 10GB database that processes 30 transactions per minute. Among these transactions, only 20% are updates. These results are averages obtained through a month.

In the laboratory scenario, Ginja has an operational cost between 62× to 222× smaller when compared with the cost of using a backup replica in a VM. The dominant factor in this scenario is the cost of uploading WAL objects to the cloud, i.e., C_{WAL_PUT}. In the hospital scenario, Ginja has a cost 14× smaller than the cost of running a backup database on a VM instance in the cloud. The benefits of Ginja in this case are not so expressive as the cost is dominated by the storage of the database.

These results show that using Ginja is substantially cheaper than maintaining VM instances in the cloud, especially for small to medium databases, which are expected to be the norm in SMEs.

Although relatively impressive, one may argue that saving $60-$270 per month is not so important, even for small companies. However, much more important than these cloud-related economical advantages, our system is arguably much simpler to manage than the alternative solution, which requires configuring a VM and a connection to the local facility (e.g, VPN, public IPs). The simplicity of operation is a key feature of our solution, which aims to make disaster recovery simpler than taking backups.

Table 2. Costs of performing cloud-based disaster recovery with AWS using Ginja or database replication with VMs. Calculated using https://calculator.s3.amazonaws.com in May 2017.

Configuration	Ginja with S3	EC2 VMs
Laboratory (10GB, 6 up/min)	**$0.42** (1 sync./m) **$1.50** (6 sync./m)	m3.medium + VPN + EBS 100IOS = **$93.4**
Hospital (1TB, 138 up/min)	**$20.3** (1 sync./m) **$21.4** (6 sync./m)	m3.large + VPN + EBS 500IOS = **$291.5**

7.3 The Cost of Recovery

The cost of recovering a database backed-up using Ginja is basically defined by the cost of downloading all DB and WAL objects. Currently, the costs of downloading one GB of data is almost 4× higher than the cost of storing it for a month in Amazon S3 [4]. Therefore, the cost of recovering a database can be approximated by $4 \times (C_{DB\_Storage} + C_{WAL\_Storage})$ plus the costs of the GET operations used to download these files (not significant). For instance, the costs of recovering from a disaster on the real clinical databases mentioned before would be $112.5 and $1.125 for the Hospital and the Laboratory, respectively. Importantly, if the database is recovered to an EC2 VM in the same location as the data, *this cost goes to zero*, as downloads from S3 to EC2 in the same region are free of charge [4].

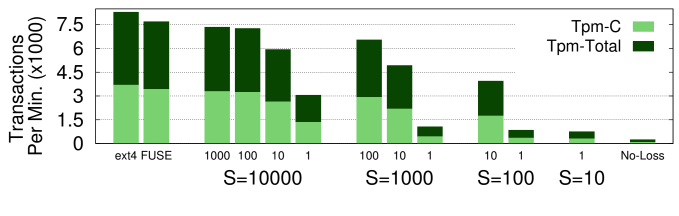
Notice that reestablishing the *local infrastructure* after a disaster using the data stored in the cloud is roughly the same in Ginja, in VM-based solutions (*Pilot Light*), and in cloud-based backups (*Backup and Restore*), as the database will need to be downloaded from the cloud.

Another important aspect of the cost of recovery is the application-specific cost of downtime. A disadvantage of Ginja when compared with a VM-based solution is the potentially slower recovery. As we shown in §8.3, recovering even a modest Ginja-backed database in a cloud VM can take few minutes. For small databases (e.g., 10GB or less) this is not problematic because starting other VMs for web and application services and configuring network services such as DNS can take roughly the same amount of time than recovering the database. This would make the RTO (Recovery-Time Objective) of a Ginja-based solution similar to a VM-based DR solution. For bigger databases, it is expected that the RTO of Ginja to be larger than the one obtained with a VM-based solution. In fact, this is the main trade-off of our design: low-cost fault-free operation potentially leads to a greater recovery time.

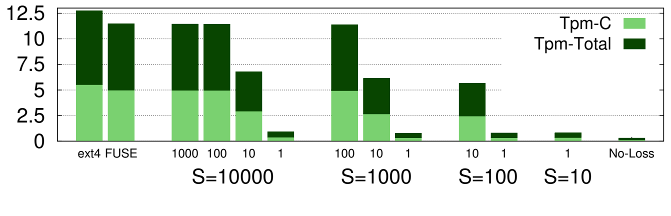
8 Experimental Evaluation

In this section we present an experimental evaluation of Ginja using PostgreSQL 9.3 [13] and MySQL 5.7 with InnoDB [10]. The objective is to assess the performance overhead due to the use of Ginja and to understand its cloud and server resources usage.

The experiments were executed in two Dell Power Edge R410 machines (one for the DBMS and Ginja and another for the benchmark software) located in our academic infrastructure, in Lisbon. Each machine has two Intel Xeon E5520 CPUs (quad-core, HT, 2.27Ghz), 32GB of RAM, and a 15k-RPMs Hard Disk Drive with 146GB. The operating system used was Ubuntu Server (14.04 LTS, 64-bits), with kernel 3.5.0-23-generic and Java 1.8.0 (64-bits). Both PostreSQL

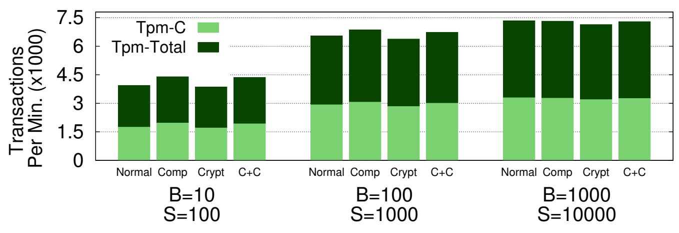


(a) PostgreSQL

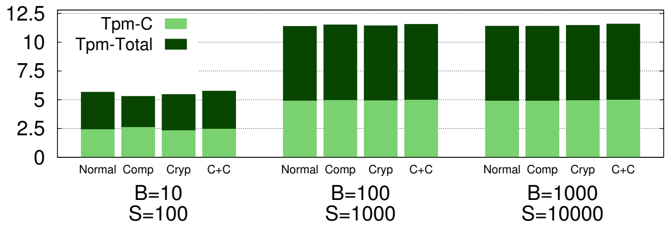


(b) MySQL

Figure 5. Influence of different configurations in the performance of GINJA with PostgreSQL and MySQL. The values of B are expressed immediately below the columns. Exceptions are the first two columns (native file system and FUSE), and the last column ($S = B = 1$).



(a) PostgreSQL



(b) MySQL

Figure 6. Effect of compression and cryptography in the performance of GINJA. The columns are grouped by configuration (B and S), and the values immediately below de columns specify whether compression, cryptography or both (C+C) are active.

and MySQL were run using their default configurations. The cloud storage service used was Amazon S3 (US East, N. Virginia).

We report average results from five executions running TPC-C [15] during five minutes. We chose this benchmark for measuring the overhead of GINJA due to its update-heavy workload (≈ 90% of updates), as our system is expected to have no impact on read-heavy workloads. For PostgreSQL, we used the BenchmarkSQL 4.1.1 tool [5] with one warehouse and five terminals, while for MySQL we used a Java implementation of TPC-C [7] configured with two warehouses and 60 terminals. We chose these configurations as they allow the DBMS to reach the highest performance without GINJA. The reported metrics are the total number of transactions per minute (Tpm-Total), and the number of *newOrder* transactions per minute while the DBMS is also processing other types of transactions (Tpm-C). In all experiments GINJA was configured with five *Uploader* threads, which corresponds to the best setup in our environment.

8.1 Performance Overhead

Performance overhead. Figure 5 shows the effect that different configurations of B (*Batch*) and S (*Safety*) have in the throughput of PostgreSQL and MySQL running TPC-C on top of GINJA. We also ran the benchmark on top of the ext4 native file system and a FUSE local file system to have baselines for comparison.

The first observation to make is that the FUSE file system presents a throughput decrease of 7% and 12% for PostgreSQL and MySQL, respectively. Since GINJA is also a FUSE file system, this will be our baseline.

The most important observation is that, for sufficiently high values of B and S, GINJA introduces a small performance loss (3.7% and 1.1% for PostgreSQL and MySQL, respectively). Furthermore, small values of B make the amount of pending updates reach S earlier, constantly blocking the DBMS and decreasing its performance.

The figure also shows results for GINJA with $S = B = 1$ (No Loss), which corresponds to synchronous replication. As expected, this configurations presents the lowest performance among the ones we tested: 248 and 348 Tpm-Total, for PostgreSQL and MySQL, respectively.

Compression and encryption. Figure 6 shows how compression and encryption influence the performance of GINJA. For PostgreSQL (Figure 6a), the use of these features made the results vary slightly, as the latency of uploading compressed data is smaller (see next section). On the other hand, encryption introduces a minimal overhead. For MySQL (Figure 6b), there are basically no changes in performance. This happens because the page size of MySQL WAL segments are quite small (512 bytes vs. 8kB in PostgreSQL), leading to diminished effects of compression and encryption in the data upload latency.

8.2 Resource Usage

Cloud usage and its implications. Table 3 shows the number of PUT operations, the size of the objects written, and the observed upload latency during the execution of the TPC-C benchmark for five minutes. We focus our discussion on PostgreSQL results, but the insights are similar for MySQL.

The results show that increasing the batch from 10 to 100 decreases the number of PUT operations performed by 80%, while an additional tenfold increase further decreases this number by almost 70%. In the same way, increasing the batch increases the object size and, consequently, the latency to write the object to the cloud. However, this increase is not linearly proportional with the increase of the object size due to coalescing of writes during the page aggregation.

Table 3. Ginja's use of storage cloud. All results are averages collected during five executions of five minutes of TPC-C for different configurations with both PostgreSQL (PG) and MySQL (MS).

Configuration	Num. PUTs (5 min)		Object Size (kB)		PUT latency (millisec.)	
	PG	MS	PG	MS	PG	MS
10/100 plain	1789	3864	386	26	692	391
10/100 C+C	1990	3994	237	11	562	376
100/1000 plain	364	1046	3018	180	2880	698
100/1000 C+C	383	1063	1908	78	2007	610
1000/10000 plain	119	139	10081	1309	7707	1552
1000/10000 C+C	119	137	6339	606	4422	1354

Table 4. Database server (eight cores with hyper-threading and 32GB of RAM) resource usage with and without Ginja.

Configuration	PostgreSQL		MySQL	
	CPU	Memory	CPU	Memory
Native FS	6.4%	4.3%	13.7%	1.2%
FUSE FS	6.9%	4.9%	14.9%	1.7%
100/1000	7.8%	6.9%	15.3%	8.1%
100/1000 Comp	11.6%	9.7%	15.8%	12.1%
100/1000 Crypt	9.1%	7.2%	16.4%	9.7%
100/1000 C+C	13.4%	9.9%	16.0%	11.1%

The table also shows that using compression (and encryption) reduces the object size by 37%, reducing the PUT latency, and bringing the benefits discussed before.

Database server resource usage. Table 4 presents the resource usage of a database server running a TPC-C workload under different configurations with and without Ginja.

For PostgreSQL, the table shows that using a Native or FUSE file system already require around 8% of the machine CPU and less than 1.6GB of memory (5%). When using Ginja, the server CPU and memory usage increase by 1% and 2%, respectively, when compared with a FUSE FS. Additionally, compression and encryption introduce some CPU load: +4.5% and +1.5%, respectively. In terms of memory, these features increase the memory usage by 3% (compression) and 0.3% (encryption). When compression and encryption are used, the overheads of these features are summed up.

For MySQL, the CPU usage is basically the same, independently of the enabled features (under the standard deviation of $\approx$ 10%). The memory usage follows the same trends as in PostgreSQL: compression demands more memory than encryption.

In the end, using Ginja with compression and encryption requires at most +7% of CPU (for PostgreSQL, which is less than a core in our server) and +10% of memory (for MySQL, less than 3.2GB) of our 8-core server with 32GB of memory. We consider these costs would not be a deterrent for using Ginja.

8.3 Recovery Time

Our last experiment measure the recovery time of Ginja after experiencing a failure when executing TPC-C for five minutes. The experiment was done with PostgreSQL, but the results for MySQL would be similar as the key factor here is the database download time from the storage service.

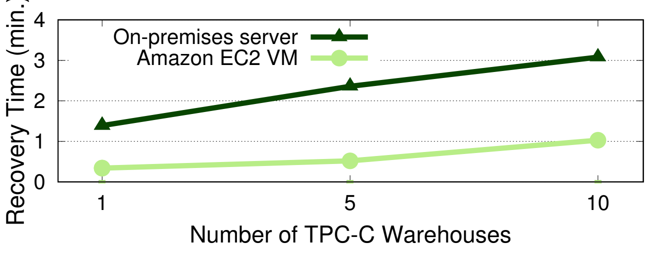


Figure 7. Recovery times of Ginja for different database sizes using a local server and a VM in the same location as the data.

We ran the experiment for three different database sizes, by varying the number of warehouses in TPC-C [15] (with a maximum database size of 1.5GB) and executed the recovery process in a machine in our infrastructure, and in an Amazon EC2 VM (located in the same region where Ginja stored the data).

As expected, the recovery time grows with the database size as more data has to be downloaded. Furthermore, the recovery time can be remarkably reduced by executing Ginja in a computing instance located in the same cloud region where the data is stored.

9 Related Work

Database replication and disaster tolerance. There is a large body of work related with database replication for fault tolerance (see [33] for a concise survey, and [23] for a criticism). For example, one of the earliest works on database disaster tolerance [34] describes a method for mirroring database nodes on a secondary, backup site. These works are mostly orthogonal to Ginja as they consider that there will be replicas to be synchronized, while we are concerned with the capture and replication of database updates on a passive remote storage in a cost-efficient way. There are database-specific middleware products such as Oracle GoldenGate [12] that capture updates from database logs and send them to a remote node or disk. However, these systems do not provide means for controlling the tradeoff between durability, costs and performance. Furthermore, they do not take into consideration the cloud storage cost and service model. Nonetheless, in the following we discuss some features provided by PostgreSQL and MySQL for implementing disaster tolerance.

PostgreSQL provides two mechanisms for helping disaster tolerance [14]. The first one, *Continuous Archiving*, consists of performing a file-system-level backup of the database directory and setting a process (the archiver) that periodically backs up completed WAL segments. This mechanism could be used to tolerate disasters by configuring the archiver process to copy the log files to a geographically remote facility such as a cloud storage service. However, the archiver process only operates over completed WAL segments, and thus it does not provide any fine-grained control over the RPO. The second mechanism is named *Streaming Replication*, and allows a primary server to replicate, in a synchronous or asynchronous way, the changes made to its database to a backup server.

MySQL also offers replication solutions very similar to PostgreSQL streaming replication, supporting asynchronous, delayed or synchronous primary-backup replication [11]. In both databases, these mechanisms could be used as a disaster tolerance solution by placing the backup replica in a cloud VM. As discussed before, this implies substantially higher costs than what we achieve with GINJA.

PostgreSQL and MySQL can also be protected by a third-party solution named *Zmanda* [17], which is a bit more closer to GINJA. This tool extends both systems backup features by allowing the specification of backup schedules and point-in-time recovery in a simple way. Zmanda also allows the execution of (full or incremental) online backups to Amazon S3 or Google Storage. Since Zmanda only backs up the state of the databases at the schedule time, *it can not provide the control over the RPO that* GINJA *provides, as it does not work at the transaction commit level*. Furthermore, being a commercial service, the costs of using Zmanda are much higher than running GINJA.

Filesystem mirroring. A common way of having disaster tolerance is by replicating data at the storage level. By continuously backing up the relevant files to remote storage facilities, a system is no longer susceptible to lose all its data if a disaster occurs in its primary infrastructure.

Two examples of such systems are *SnapMirror* [39] and *Seneca* [31]. Both use asynchronous replication in order to avoid any significant loss of performance. The main difference between these two solutions is that the first replicates consistent file system snapshots, while the second sends batches of updates to the remote site.

The most important advantage of such solutions is that they allow any application to protect its data, without requiring changes to its source code. However, they do not consider the semantics of the applications, which can result in inconsistent states after recovery. Additionally, these solutions require computing instances running on the backup site, which implies higher costs than running GINJA.

Virtual machine replication. The virtualization of IT resources is one of the key features of modern DR strategies [41, 49]. Here we discuss some works for transparent VM replication that could be used for disaster recovery in database systems.

RemusDB [36] is an extension for the Remus VM replication system [26] that provides high availability for DBMS in a transparent way. This is achieved by running the DBMS in a virtual machine, making the virtualization layer perform the tasks related with data replication, failure detection, and recovery. A key limitation of RemusDB is that it was not designed for wide-area replication, and the higher latencies can render the system impractical.

SecondSite [40] is another extension for Remus, specifically designed for disaster recovery. The system continuously replicates the entire state of virtual machines in a primary site to backup VMs in a different geographic location, which can transparently assume the responsibility of providing the service if a disaster strikes. SecondSite deals with the limitations of wide-area replication by making a better use of bandwidth through checkpoint compression, and by using quorums of servers for detecting failures.

PipeCloud [50] is a cloud-based disaster recovery system for multi-tier client-server applications running on a set of VMs. This system runs in the virtual machine monitor of each cloud physical server and replicates all disk writes to geographically distant backup servers.

All these virtualization-based approaches have the advantage of performing fast failover since they include a backup VM running in a secondary site ready to take over when a disaster is detected in the primary infrastructure. Such additional computing resources implies higher operational and management costs.

Cloud-backed storage services. Although the following solutions were not explicitly conceived for disaster recovery, the mechanisms they employ are often similar to the ones we use in GINJA.

Brantner et al. [21] presents a DBMS core that uses Amazon S3 as its storage subsystem. This core allows retrieving pages from S3, buffering them locally (in memory or disk), updating them, and writing them back to the cloud. All cloud operations are coordinated by a page manager, on top of which there is a record manager that provides a record-oriented interface to the applications. The work proposes several protocols for accessing cloud services with different guarantees, but its design does not prioritizes either cost or performance. Furthermore, integrating this solution with existing DBMS requires a substantial reengineering effort, on the contrary of GINJA.

Cumulus [47] is a utility that performs efficient file system backups to cloud storage services. Thus, it can be used to take snapshots of the data directory where DBMS writes preventing the failure of the local infrastructure.

Cloud-backed file systems such as *BlueSky* [48] and *SCFS* [20] translate local file system operations to a cloud storage service with a minimum or no use of cloud VMs. SCFS in particular provides strong consistency and durability guarantees and thus could be used to provide disaster recovery for a database running on top of it. However, the system implements only synchronous or asynchronous replication of whole files, which means that the database files replication will be very inefficient.

10 Conclusion

We presented GINJA, a transactional DBMS disaster recovery middleware that uses public cloud storage services for offering efficient and low-cost DR. Our current prototype supports PostgreSQL and MySQL, and the experimental results show that using our system degrades the database performance by less than 5% when running TPC-C, with less than 10% additional CPU and memory load on our server. Furthermore, GINJA is between 14× to 222× cheaper than having a VM-based cloud disaster recovery service for a database used in a real application.

Acknowledgments. We thank Paulo Sousa from MaxData Software for providing statistics about the laboratory and hospital databases. This work was supported by FCT through projects LaSIGE (UID/CEC/00408/2013) and IRCoC (PTDC/EEI-SCR/6970/2014), and by the European Commission through the H2020 SUPERCLOUD project (643964).

References

[1] 2016. Business continuity trends and challenges 2016. (Jan. 2016). http://www.continuitycentral.com/index.php/news/business-continuity-news/776-business-continuity-trends-and-challenges-2016.
[2] 2017. Amazon EC2 Instance Types. (2017). https://aws.amazon.com/ec2/instance-types/.
[3] 2017. Amazon RDS Multi-AZ Deployments. (2017). https://aws.amazon.com/rds/details/multi-az/.
[4] 2017. Amazon S3 pricing. (2017). https://aws.amazon.com/s3/pricing/.
[5] 2017. BenchmarkSQL. (2017). https://bitbucket.org/openscg/benchmarksql.
[6] 2017. FUSE-J. (2017). http://fuse-j.sourceforge.net/.
[7] 2017. Java TPC-C. (2017). https://github.com/AgilData/tpcc.
[8] 2017. Microsoft Azure Site Recovery. (2017). https://azure.microsoft.com/en-us/services/site-recovery/.
[9] 2017. MySQL - The InnoDB Storage Engine. (2017). http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.
[10] 2017. MySQL 5.7 Documentation. (2017). http://dev.mysql.com/doc/refman/5.7/en/.

[11] 2017. MySQL Replication. (2017). http://dev.mysql.com/doc/refman/5.7/en/replication.html.

[12] 2017. Oracle GoldenGate. (2017). http://www.oracle.com/technetwork/middleware/goldengate/overview/

[13] 2017. PostgreSQL. (2017). http://www.postgresql.org/.

[14] 2017. PostgreSQL Documentation. (2017). http://www.postgresql.org/docs/.

[15] 2017. TPC-C Benchmark. (2017). http://www.tpc.org/tpcc/.

[16] 2017. VMware vCloud Air Disaster Recovery. (2017). https://www.vmware.com/cloud-services/infrastructure/vcloud-air-disaster-recovery.

[17] 2017. Zmanda Recovery Manager for MySQL. (2017). http://www.zmanda.com/.

[18] BBC News. 2017. WannaCry ransomware cyber-attacks slow but fears remain. (May 2017). http://www.bbc.com/news/technology-39920141.

[19] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage* 9, 4 (2013).

[20] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. 2014. SCFS: a shared cloud-backed file system. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*.

[21] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD'08)*.

[22] Peter Brouwer. 2011. The Art of Data Replication. (2011). Oracle Technical White Paper.

[23] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. 2008. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD'08)*.

[24] Rafal Cegiela. 2006. Selecting technology for disaster recovery. In *International Conference on Dependability of Computer Systems (DepCos-RELCOMEX'06)*.

[25] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. 2013. From ARIES to MARS: Transaction Support for Next-generation, Solid-State Drives. In *Proceedings of 24th ACM/SIGOPS Symposium on Operating Systems Principles (SOSP'13)*.

[26] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*.

[27] Sharon Fisher. 2014. On the Quest for the Mysterious Source of the "Data Loss Causes Company Failure" Statistic. (Feb. 2014). http://itknowledgeexchange.techtarget.com/storage-disaster-recovery/on-the-quest-for-the-mysterious-source-of-the-data-loss-causes-company-failure-statistic/.

[28] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. 2016. Why does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*.

[29] Pedro Hernandez. 2014. Small Business IT Survey: No Backup, No Data, No Business. (May 2014). http://www.smallbusinesscomputing.com/biztools/small-business-it-survey-no-backup-no-data-no-business.html.

[30] B. Hou, F. Chen, Z. Ou, R. Wang, and M. Mesnier. 2016. Understanding I/O Performance Behaviors of Cloud Storage from a Client's Perspective. In *Proceedings of the 32th IEEE International Conference on Massive Storage Systems and Technology (MSST'16)*.

[31] Minwen Ji, Alistair C Veitch, John Wilkes, and others. 2003. Seneca: remote mirroring done write. In *Proceedings of the 2003 USENIX Annual Technical Conference (ATC'03)*.

[32] Kimberly Keeton, Cipriano A Santos, Dirk Beyer, Jeffrey S Chase, and John Wilkes. 2004. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*.

[33] Bettina Kemme, Ricardo Jimenez-Peris, and Marta Patiño-Martínez. 2010. *Database Replication*. Morgan & Claypool.

[34] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. 1991. Management of a Remote Backup Copy for Disaster Recovery. *ACM Transactions on Database Systems* 16, 2 (1991).

[35] Edward Kovacs. 2014. Downtime and Data Loss Cost Enterprises $1.7 Trillion Per Year: EMC. (Dec. 2014). http://www.securityweek.com/downtime-and-data-loss-cost-enterprises-17-trillion-year-emc.

[36] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. 2013. RemusDB: Transparent high availability for database systems. *The VLDB Journal* 22, 1 (2013).

[37] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems* 17, 1 (1992).

[38] Tiago Oliveira, Ricardo Mendes, and Alysson Bessani. 2016. Exploring Key-Value Stores in Multi-Writer Byzantine-Resilient Register Emulations. In *Proceedings of the 20th International Conference On Principles Of DIstributed Systems (OPODIS'16)*.

[39] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. 2002. SnapMirror: file system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*.

[40] Shriram Rajagopalan, Brendan Cully, Ryan O'Connor, and Andrew Warfield. 2012. SecondSite: disaster tolerance as a service. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE'12)*.

[41] Glen Robinson, Attila Narin, and Chris Elleman. 2014. Using Amazon Web Services for Disaster Recovery. (Dec. 2014). Amazon Web Services white paper.

[42] Susan Snedaker. 2013. *Business continuity and disaster recovery planning for IT professionals*. Newnes.

[43] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data (SIGMOD'86)*.

[44] Symantec. 2009. SMB (Small and Medium Business) security and data protection: survey shows high concern, less action. (2009). White paper: SMB Survey.

[45] Symantec. 2016. Ransomware and Business 2016. (2016). ISTR Special Report.

[46] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. 2015. Terra Incognita: On the Practicality of User-Space File Systems. In *Proceedings of the 7th USENIX workshop on hot topics in Storage and File Systems (HotStorage'15)*.

[47] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. 2009. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage* 5, 4 (2009).

[48] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. 2012. BlueSky: A Cloud-backed File System for the Enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

[49] Timothy Wood, Emmanuel Cecchet, KK Ramakrishnan, Prashant Shenoy, Jacobus Van Der Merwe, and Arun Venkataramani. 2010. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *Proceedings of the 1st USENIX workshop on hot topics in cloud computing (HotCloud'10)*.

[50] Timothy Wood, H Andrés Lagar-Cavilla, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. 2011. PipeCloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)*.