UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

LOW-COST CLOUD-BASED DISASTER RECOVERY FOR TRANSACTIONAL DATABASES

MESTRADO EM ENGENHARIA INFORMÁTICA
Arquitetura, Sistemas e Redes de Computadores

Joel Melão Alcântara

Dissertação orientada por:
Professor Alysson Neves Bessani

2016

Agradecimentos

Em primeiro lugar agradeço aos meus pais e ao meu irmão, por tudo o que fizeram e continuam a fazer por mim. Todo o apoio, carinho e educação que me proporcionam fez de mim quem eu sou hoje, e eu estarei para sempre grato por isso.

Agradeço à minha namorada Maria, por estar sempre lá para mim nos bons e maus momentos, tornando a minha vida mais feliz.

Aos grandes amigos que me acompanharam neste percurso pela Faculdade de Ciências da Universidade Lisboa: Filipe Custódio, Frederico Brito, João Rebelo, José Soares e Luís Ferrolho. Espero nunca vos perder de vista. Um especial obrigado ao José Soares, que tanto me ajudou nos primeiros tempos de faculdade, quando a informática parecia ser difícil demais.

Ao meu orientador – Professor Alysson Neves Bessani – por todo o apoio e bom ambiente que me proporcionou ao longo deste projeto. Aos restantes professores da faculdade que tanto me ensinaram durantes os últimos 5 anos da minha vida. Agradeço também ao Tiago Oliveira e ao Ricardo Mendes pelo apoio prático que me deram durante o desenvolvimento deste projeto. Por fim, agradeço à Faculdade de Ciências da Universidade de Lisboa, por todas as condições oferecidas durante o meu percurso académico.

Este trabalho foi suportado pela Comissão Europeia através do projeto SUPERCLOUD (H2020/ICT-643964), e pela Fundação para a Ciência e a Tecnologia (FCT) através de seu programa multianual (LaSIGE).

Aos meus pais Arlinda e António,
Ao meu irmão André.

Resumo

A fiabilidade dos sistemas informáticos é uma preocupação fundamental em qualquer organização que dependa das suas infraestruturas de tecnologias de informação. Particularmente, a ocorrência de desastres introduz sérios obstáculos à continuidade de negócio. Ao contrário das falhas individuais de componentes, os desastres tendem a afetar toda a infraestrutura que suporta o sistema [1]. Consequentemente, a aplicação de técnicas de recuperação de desastres (*Disaster Recovery* ou DR) é crucial para assegurar alta disponibilidade e proteção de dados em sistemas de informação.

As estratégias tradicionais para recuperação de desastres baseiam-se na realização periódica de cópias de segurança utilizando dispositivos de armazenamento em fita, que são armazenados numa localização distante (de modo a não serem suscetíveis aos mesmos desastres que a infraestrutura do sistema). As abordagens mais recentes, por sua vez, passam por replicar os recursos computacionais que compõem o sistema numa infraestrutura remota que pode ser utilizada para dar continuidade ao serviço em caso de desastre. Mais uma vez, a distancia geográfica entre as infraestruturas deve ser tão grande quanto possível.

Tendo em conta estes requisitos, as *clouds* publicas surgem como uma excelente oportunidade para a concretização de sistemas de recuperação de desastres [2]. A elasticidade da *cloud* elimina a necessidade da replicação completa do serviço na infraestrutura secundária, permitindo que apenas os serviços mínimos sejam executados na ausência de falhas, e que os custos operacionais do sistema sejam pagos apenas em caso de necessidade, i.e., aquando da ocorrência de desastres. Isto possibilita ganhos substanciais quando comparado com os custos fixos (e.g., hardware, gestão, energia, conectividade) de uma infraestrutura dedicada [3].

A criação de uma estratégia de recuperação de desastres na *cloud* requer a definição de um conjunto de instancias computacionais a executar os serviços sem estado que compõem o serviço (e.g., servidores *web*, *middleboxes*, servidores de aplicação) e um outro conjunto de instâncias a executar os componentes do sistema com estado persistente, normalmente composto por um sistema de gestão de bases de dados (SGBD). Nas soluções atuais para recuperação de desastres na *cloud*, os serviços sem estado permanecem inativos durante a operação normal, enquanto os serviços com estado são mantidos

em execução, mas em modo passivo, apenas recebendo atualizações das suas cópias presentes na infraestrutura primária. Estas atualizações podem ser concretizadas através da replicação oferecida pelos próprios SGBDs ou por funcionalidades concretizadas ao nível do sistema operativo ou da camada de virtualização [4–6].

Neste trabalho propomos o GINJA, um sistema de recuperação de desastres que recorre exclusivamente a serviços de armazenamento na *cloud* para replicar uma importante classe de sistemas – os sistemas de gestão de bases de dados. O GINJA atinge três principais objetivos que tornam a proposta inovadora: reduzir os custos da recuperação de desastres; permitir um controlo preciso sobre os compromissos de custo, durabilidade e desempenho; e adicionar um *overhead* mínimo ao desempenho do SGBD.

O principal fator que permite ao GINJA reduzir custos prende-se com o facto de este ser completamente centrado no uso de serviços de armazenamento da *cloud* (e.g., Amazon S3, Azure Blob Storage). Esta decisão elimina a necessidade de manter máquinas virtuais em execução na *cloud* para receber atualizações da infraestrutura primária, o que resultaria em elevados custos monetários e de manutenção. Deste modo, o GINJA define um modelo de dados (concebido especificamente com o intuito de reduzir custos monetários e permitir uma realização eficiente de backups), e sincroniza para a *cloud* os dados gerados pelo SGBD de acordo com esse modelo. Em caso de desastre, uma instancia computacional é executada em modo de recuperação com o fim de descarregar os dados armazenados na *cloud* e dar continuidade ao serviço. É de referir que o tempo de recuperação pode ser reduzido drasticamente se o processo de recuperação for executado em recursos computacionais presentes na infraestrutura utilizada para armazenar dados (e.g., Amazon EC2, Azure VM).

A execução do GINJA é baseada na configuração de dois parâmetros fundamentais: B e S. B define o número de alterações nas bases de dados que são incluídas em cada sincronização com a *cloud*, pelo que tem efeitos diretos no custo monetário (dado que cada carregamento de dados para a *cloud* tem um custo associado). S define o número máximo de operações de escrita no SGBD que podem ser perdidas em caso de desastre. De modo a garantir que nunca são perdidas mais do que S operações nas bases de dados, o GINJA bloqueia o SGBD sempre que necessário. Por consequência, este parâmetro relaciona-se diretamente com o desempenho do sistema. Conjuntamente, os parâmetros B e S fornecem aos nossos clientes um controlo preciso relativamente ao custo e durabilidade do SGBD no qual o GINJA é integrado.

O GINJA foi implementado sob a forma de um sistema de ficheiros ao nível do utilizador [7], que intercepta as chamadas ao sistema de ficheiros efetuadas pelo SGBD e realiza sincronizações com a *cloud* de acordo com os parâmetros B e S. Esta decisão torna o GINJA numa solução bastante portável, dado que não necessita que sejam efetuadas quaisquer alterações ao SGBD, e permite que sejam criadas extensões com o fim de suportar outros sistemas de gestão de bases de dados.

Neste projeto realizamos também uma avaliação extensiva ao nosso sistema, que analisa tópicos como custos monetários, eficiência e utilização de recursos. Os resultados obtidos ilustram os compromissos fundamentais de custo, desempenho e limite de perda de dados (i.e., durabilidade em caso de desastre), e mostram que a utilização do GINJA leva a uma perda de desempenho negligenciável em configurações nas quais alguma perda de dados é aceitável.

O trabalho desenvolvido neste projeto resultou na publicação: Joel Alcântara, Tiago Oliveira, Alysson Bessani; *Ginja: Recuperação de Desastres de Baixo Custo para Sistemas de Gestão de Bases de Dados*, no INForum 2016, na *track* "Computação Paralela, Distribuída e de Larga Escala" [8]. Além disso, o software desenvolvido será utilizado na demonstração do projeto H2020 SUPERCLOUD, juntamente com o sistema de gestão de análises clínicas da MAXDATA, a ser realizada na avaliação intermédia do projeto, em meados de Setembro.

Palavras-chave: Recuperação de Desastres, Tolerância a Faltas, Computação na Cloud, Sistemas de Gestão de Bases de Dados, Replicação

Abstract

Disaster recovery is a crucial feature to ensure high availability and data protection in modern information systems. The most common approach today consists of replicating the services that make up the system in a set of virtual machines located in a geographically distant public cloud infrastructure. These computational instances are kept executing in passive mode, receiving updates from the primary infrastructure, in order to remain up to date and ready to perform failover if a disaster occurs at the primary infrastructure. This approach leads to expressive monetary and management costs for keeping virtual machines executing in the cloud.

In this work, we present GINJA – a disaster recovery system for transactional database management systems that relies exclusively on public cloud storage services (e.g., Amazon S3, Azure Blob Storage) to backup its data. By eliminating the need to keep servers running on a secondary site, GINJA reduces substantially the monetary and management costs of the disaster recovery. Furthermore, our solution also includes a configuration model that allows users to have a precise control about the cost, durability and performance trade-offs, and introduces a minimum overhead to the performance of the database management system.

Additionally, GINJA is implemented as a specialized file system in user space, which brings major benefits in terms of portability, and allows it to be easily extended to support other database management systems.

Lastly, we have performed an extensive evaluation of our system, that covers aspects such as performance, resource usage and monetary costs. The results show that GINJA is capable of performing disaster recovery with small monetary costs (less than 5 dollars for certain practical configurations), while introducing a minimum overhead to the database management system (12% overhead for the TPC-C workloads with at most 20 seconds of data loss in case of disasters).

Keywords: Disaster Recovery, Fault-Tolerance, Cloud Computing, Database Management Systems, Replication.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems have a huge role in modern society. Not only ordinary services are increasingly adopting digital solutions, as new IT services and business models are emerging. Such systems have very strict data protection and high availability (HA) requirements as, generally, even partial data loss or short periods of downtime can lead to severe consequences (such as financial loss). Such systems have very strict data protection and high availability (HA) requirements since, generally, even short periods of downtime can lead to severe consequences (such as financial loss).

Achieving high availability and data protection is not an easy task. It involves applying fault tolerance techniques in order to ensure that the system continues to perform its functions despite the occurrence of failures. Such failures can have several sources such as hardware failures, power outages or disasters.

The occurrence of disasters introduces some serious challenges to the task of achieving high availability and data protection. 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, causing a greater damage to the service provided. Consequently, the ability to tolerate disasters requires a careful planning [1].

In order to tolerate disasters, a system must have backup resources placed in a remote site that is not vulnerable to the same disasters as the primary infrastructure. Such resources, used to replicate the state of the system and perform failover, result in additional costs for the enterprises that host IT services. Using cloud computing for integrating disaster recovery solutions is an excellent way of reducing costs [3]. Its pay-as-you-go model, available on demand resources and high degree of automation are extremely attractive features for disaster recovery systems. Additionally, the fact that the cloud providers carefully manage and maintain their infrastructures lowers the management efforts of hosting our backup resources in the cloud.

Cloud providers offer a broad range of different services. Consequently, a variety of cloud-based disaster recovery solutions exists today, differing in the services they use, the layer at which they perform DR and the requirements they focus on.

1.1 Motivation

Unfortunately, data loss is currently a common event that may lead to disastrous consequences. Although statistics about data losses and its effects are sometimes misleading [9, 10], recent surveys showed that data loss costs $1.7 Trillion per year for medium and big companies [11]. For small businesses the situation might be even worse. Few years ago a survey by Symantec showed that 40% of small and medium companies do not do regular backups [12]. We believe that this situation improved in the last years, but it is unlikely that this protection gap disappeared. A more recent survey showed that 58% of small and medium companies could not sustain any amount of data loss [13]. However, the same survey shows that 62% of these companies does not backup their data on a daily basis. These numbers clearly show that even simple backup routines are still a challenge for small and medium companies, and demonstrate the fact that fully automated disaster recovery are not yet widely deployed. Lack of budget and automation are usually pointed as key challenges for implementing effective business continuity plans [14].

Traditional disaster recovery approaches range from periodic tape backups kept in remote sites, to continuous replication of data to distant secondary datacenters [6]. Such solutions usually require expensive facilities (such as a secondary infrastructure and high bandwidth links connecting both sites) that are only used in the unlikely event of a disaster.

Fortunately, the emergence of cloud computing allowed the creation of low-budget disaster recovery solutions [3]. The enterprises no longer need to invest in redundant infrastructures to ensure high availability and data protection during disasters. Instead, they rely on cloud services to host a portion of their system and, in the occurrence of a disaster, they can quickly initiate more resources.

Cloud based disaster recovery solutions vary on many levels. First, it is possible to implement it in different layers such as within the application [15], in the virtualization platforms [5, 6] or in the filesystem or block device levels [16, 17]. Additionally, we can rely on different cloud services to host our DR resources (virtual machine and storage instances are the most common approaches) [2]. Lastly, there are several replication techniques for propagating the application state to the backup site: some systems apply synchronous replication to protect data, whereas others prefer to ensure higher performance during normal operation and opt by an asynchronous scheme [6].

Despite the existing variety of disaster recovery approaches, we consider that there is still room for innovation in this area. We believe that cloud services have the potential of reducing costs even more for certain disaster recovery scenarios if properly used. Furthermore, we argue that allowing the users to decide the balance between performance and data loss is necessary to create a flexible DR solution.

1.2 Objectives

In this project, we aim to create a low cost cloud based disaster recovery solution. Since most of the systems rely on DBMS to store and manage its data, we have decided to focus our work in these systems. Specifically, our solution will be integrated in the file system layer, and will rely on cloud storage services to protect the database information from disasters.

We focus on four main objectives for our disaster recovery system.

First, we want to leverage the cloud storage pricing model to create an extremely low-cost disaster recovery solution. By studying how cloud providers charge for their storage services (which are already cheap) and understanding how DBMS manage their data, we can optimize our backups to be as cheap as possible.

Second, our solution reduces substantially the operational costs of deploying and maintaining a disaster recovery system. By using storage resources rather than computing services, our system automates the integration and management of the cloud resources that compose the secondary infrastructure.

The third objective of this work is to provide its users with a fine-grained control over the data that is protected from disasters. We believe that this feature is important because, unlike us, the users can take decisions based on how critical their data is and how they are willing to sacrifice data protection to achieve better performance (or the other way around). By allowing the users to configure our system according with their requirements, we can build a more flexible solution.

Lastly, we intent to optimize our service so that it adds the less overhead possible to the application it protects under failure-free operation. Obviously, the performance impact that our system causes depends on its configurations. Nevertheless, it must be the least intrusive possible in order to be an attractive solution for most applications.

It should be noted that minimizing the recovery time is not one of our primary goals. This means that this factor will not be decisive during he design of our solution. Nevertheless, we will to take this element into account as much as possible, as long as it does not have a negative impact on our main objectives.

1.3 Contributions

In this work we have devised GINJA: a solution that relies in cloud storage services to enable database management systems to recover from disasters. The main features that distinguish our work from the existing DR approaches are: low monetary costs, high degree of automation, fine grained control over data loss in the event of a disaster, and low performance overhead during failure-free execution.

Additionally, we have implemented a prototype of GINJA to prove the feasibility of our solution. Although our present implementation only provides disaster recovery to

PostgreSQL [18], it was specifically designed in a modular way in order to be easily extended to support other database management systems.[1]

Finally, we have also performed an evaluation of GINJA both in terms of performance and monetary costs. The results suggest that our proposal is capable of performing disaster recovery in a cheap and efficient way, while allowing our users to have a strict tight control over the data that can be lost when a disaster occurs.

1.4 Publications and Exploitation

The work described in this dissertation resulted in the publication of a paper in INForum 2016, in the track "Parallel, distributed and large scale computing" [8]. Furthermore, GINJA will be a key demonstration in the intermediate review of the SUPERCLOUD H2020 project. In this demonstration, the system will be integrated with the MAXDATA clinical software.

1.5 Planning

The Gantt chart present in Figure 1.1 illustrates the project schedule of this dissertation.

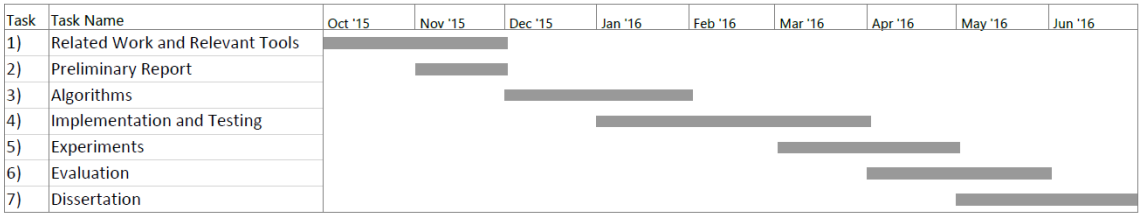

Figure 1.1: Work plan.

Let us now provide a brief description of each task that compose this project:

- **Task 1 (October and November of 2015):** Study the related work and learn to use the tools required to this project.
- **Task 2 (November of 2015):** Write the preliminary report.
- **Task 3 (December of 2015 and January of 2016):** Devise and analyse the algorithms necessary to implement GINJA.
- **Task 4 (January, February and March of 2016):** Implement and test GINJA.
- **Task 5 (March and April of 2016):** Conduce experiments on GINJA.

[1] In fact, a researcher at LaSIGE is currently developing a module that enables our system to perform disaster recovery for MySQL [19]

- **Task 6 (April and May of 2016):** Analyse and evaluate GINJA based on the results collected in the previous task.
- **Task 7 (May and June of 2016):** Write the dissertation.

The last tasks present in this schedule were slightly delayed by the iterative process of optimizing and evaluating our algorithms.

1.6 Document Organization

The remaining of this document is organized in the following way:

- Chapter 2 – covers the works that are related to this project. The two initial subsections of this chapter expose the main disaster recovery concepts and how cloud services can be used to create DR solutions. The remaining subsections distinguish different approaches for tolerating disasters and cover relevant works in the field.
- Chapter 3 – presents the I/O in Database Systems. In this chapter we start by covering the database concepts that are relevant to our work, then we include a long section that describes the internals of PostgreSQL.
- Chapter 4 – an in-depth description of the solution we have devised in this project, as well as the reasoning behind our design decisions. We start by exposing the principles and assumptions and the general architecture of our solution. Then we present our configuration parameters. After that we explain the data model our system uses to manage data in the cloud. At last we explain the algorithms employed by our system to perform disaster recovery.
- Chapter 5 – covers the most relevant technical details related to the implementation of GINJA. Such details include the architecture of our software system, as well as its integration with the database management system.
- Chapter 6 – includes an evaluation of our solution. We start by presenting an evaluation of the monetary costs inherent in the usage of our system, and then we present and discuss the results of our practical experiments.
- Chapter 7 – summarizes the conclusions of this project.

Chapter 2

Related Work

This chapter includes a set of systems and approaches that are somehow related to this work. In the first two sections we introduce the most important disaster recovery concepts and explore how cloud services can be used to implement disaster recovery. Then, we present solutions that provide high availability and disaster recovery services at the virtualization level (Section 2.3) and at the file system level (Section 2.4). Afterwards, in Sections 2.5 and 2.6, we explore how other types of solutions (specifically DBMS recovery mechanisms and systems that rely on cloud storage services to store data) can be used to achieve disaster recovery. Finally in Section 2.7 we conclude with a discussion of the disadvantages and benefits of the solutions previously described.

2.1 Disaster Recovery Concepts

A *Disaster* is any event that has a negative impact on a company's business continuity or finances [2]. Examples of disasters include network and power outages, hurricanes, earthquakes, floods, and so forth. *Disaster Recovery* (DR) is the area that allows IT systems to tolerate or recover from the damage caused by disasters. Specifically, DR solutions aim to protect application data from being lost and minimize de downtime caused by disasters.

The infrastructure used to provide services to users during normal operation is located in the *Primary Site* [20]. Since disasters can affect wide areas, the only way to tolerate them is by having our data replicated in a geographically distant location that is not vulnerable to the same disasters. This alternative location is called *Secondary Site* (or Backup Site).

There are several ways of performing disaster recovery [1]. The most traditional approach for disaster recovery is *Tape Backup and Restore* [21]. This technique consists of periodically taking consistent snapshots of the data (optionally interspersed with incremental backups), storing it into tape drives and sending those tapes offsite. When a disaster occurs, the most recent data backups are loaded to a new server that hosts the system from that point onward. Although this approach is attractive for being low-cost,

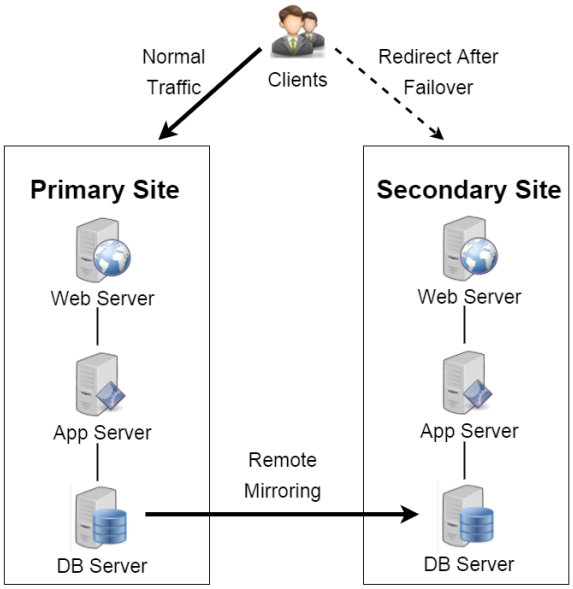

Figure 2.1: Architecture of a remote mirroring DR system.

it has the disadvantages of having long recovery and restoring the systems to an outdated state (because the backup intervals are typically long). An alternative strategy to perform disaster recovery is *Remote Mirroring* [17]. In this approach, the system continuously replicates its data to an online remote mirror placed in a secondary site. If a disaster occurs in the primary site, the resources in the secondary site can be configured to become primary. Despite being expensive, this DR technique is favourable because it allows IT systems to tolerate disasters, while reducing substantially the recovery time. The general architecture of a Remote Mirroring disaster recovery system is represented in Figure 2.1.

The data replication between sites can be performed essentially in two ways: synchronously or asynchronously [6, 22]. In *Synchronous Replication* the primary site can only return successfully from a write operation after it has been acknowledged by the secondary site. Although this replication scheme guarantees that no data is ever lost, it introduces a high performance overhead to the system being replicated, specially when there is a long distance between the primary and secondary sites (which is desirable in DR solutions). In *Asynchronous Replication* the primary site is allowed to proceed its execution without waiting for the replication to be completed at the secondary site. This type of replication overcomes the performance limitations of synchronous replication at the expense of allowing data to be lost if a failure occurs.

Besides managing state replication, a disaster recovery solution must also be able to perform failover and failback [3]. *Failover* consists of detecting when the primary infrastructure is down and activating the backup site as primary in order to guarantee system continuity. When the disaster terminates, the control of the system must be reverted to its original site. This procedure is called *Failback*.

In order to tolerate disasters a system has to have extra resources (such as a secondary infrastructure and high bandwidth links between sites) which involves monetary costs. Furthermore, there are several requirements for disaster recovery that vary from system to system [16]. Such requirements include: recovery time, consistency degree of the data recovered, performance impact during normal operation, distance between sites and costs. For this reason, it is very important to plan a disaster recovery solution suitable for each system.

There are two time parameters that need to be defined during DR planning [20]. *Recovery Point Objective* (RPO) is the number of updates (in terms of time) that can be lost due to a disaster. *Recovery Time Objective* (RTO) refers to the amount of downtime that is acceptable before a system recovers from a disaster. Although the definition of RTO and RPO is hard for most systems, there are variables that help establishing these time parameters. Examples of such variables are: the type of the computer system we intend to protect (e.g., real-time control systems typically need tighter RTO and RPO values than back-office applications) and how fast and severe are the impacts of a disaster in a system (e.g., one hour outages have different impacts on different applications).

2.2 Cloud-based Disaster Recovery

Cloud computing is extremely attractive for disaster recovery solutions. The use of cloud infrastructures for disaster recovery can lower costs (due to its pay-as-you-go model) and reduce recovery times (since cloud resources are available on demand and can be automatically allocated).

Public clouds provide us with a set of basic services, such as storage, computing, networking, database, deployment and security services that are available in various regions so the user can choose the most appropriate. This variety of services are building blocks from which we can build disaster recovery solutions suitable for each situation (taking into account aspects such as our objectives and budget) [2]. Here are some examples of possible cloud based DR solutions:

- Storing data backups in a cloud storage object and, if a disaster occurs, the backed up data can be transferred to an alternative site or to a cloud computing instance in order to ensure availability. The data transfer to and from the storage object can be performed over the internet or using cloud networking services (for instance with increased bandwidth throughput).

- Having a minimal version of our environment (the most critical core elements) executing in the cloud and, when a failure occurs, provision the rest of the infrastructure around that core in order to restore the complete system. The remainder of the infrastructure can be quickly provisioned if we have our preconfigured virtual machine images ready to be started.

These were only two possible approaches to perform disaster recovery using general cloud services. The second solution achieves a quicker recovery time than the first one, since some pieces of the system are already running in the cloud when a disaster occurs, but it is also more expensive.

Some public clouds also provide specific disaster recovery services that typically use their infrastructures as a secondary site. Examples of such services are Azure Site Recovery [23] and vCloud Air Disaster Recovery [24].

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 [3].

The more cloud resources a system requires in failure-free operation, the higher will be the costs of the DR solutions. Even if the costs during failover are slightly higher in cloud based solutions, the overall costs can still be smaller since failover is supposed to be seldom performed.

2.3 Virtual Machine Replication

The virtualization of IT resources has the potential of decreasing recovery times and simplifying the management of DR solutions. Virtualization features such as hardware independency and ease of automating tasks (such as backing up and restoring the state of a VM) led to the development of numerous VM based disaster recovery systems.

Remus [25] is a system that provides high availability as a service for unmodified applications running within virtual machines (on the Xen hypervisor) in commodity hardware. In this system, the primary VM issues fine-grained checkpoints of its entire state (including CPU, memory, disk and network device state) and replicates them asynchronously to a backup host, while executing speculatively between checkpoints. On the event of failure of the primary host, the backup VM transparently resumes the execution from the last valid checkpoint with only seconds of downtime (in a local network). Remus ensures that no externally visible state is ever lost since it is never exposed - the outgoing packets generated speculatively by the primary VM are only sent to the client upon the completion of the next checkpoint.

RemusDB [4] is a Remus based system that provides high availability for database management systems (DBMS) in a transparent manner. In this solution, the DBMS is executed in a virtual machine and the virtualization layer is responsible for performing the high availability tasks such as capturing the state of primary VM's and disseminating it to the backup VM, detect failures and perform failover.

The authors found that the use of Remus on DBMS introduces a significant performance overhead due to the facts that database systems use memory intensively and that their workloads are sensitive to network latency, and designed solutions for that problem.

These solutions include facilities that reduce the latency on the client-server communication, and a DBMS aware checkpointing system that reduces the volume of data transferred during checkpoints. RemusDB provides a fast failover with low performance overhead that preserves full ACID transactional guarantees.

SecondSite [5] is a Remus-based disaster recovery service for virtual machines running in cloud environments. This system continuously replicates the entire state of several virtual machines to backup images in a different geographic location. If the primary site fails, the backup site is capable of reconfiguring the network and resuming the execution of the protected virtual machines from the last consistent checkpoint in a completely transparent manner.

Although SecondSite's general idea is similar to Remus, the latter was designed for operating in Local Area Networks (LAN) whereas SecondSite operates in a Wide Area Network (WAN) environment. This allows SecondSite to tolerate disasters, but introduces new challenges such as network constraints (lower bandwidth and higher latency), and harder failure detection and recovery. The authors coped with these challenges by implementing a more efficient use of bandwidth (this includes techniques like checkpoint compression), placing quorum servers in a different network that act as arbitrators during failure detection, leveraging BGP multi-homing to achieve failure recovery, and building a resynchronization module that synchronizes the storage between sites after a crashed site comes back online.

It should be noted that, like Remus, this system buffers the outgoing packets generated by the primary server until the checkpoints are acknowledged by the backup server. However, in SecondSite the backup server is located in a remote site. The fact that this system operates in a WAN environment increases the protected server's response time.

PipeCloud [6] is a cloud-based disaster recovery system for client server applications. This system runs in the virtual machine manager of each physical server and replicates all disk writes to geographically distant backup servers.

PipeCloud uses a replication scheme called Pipelined Synchronous Replication that combines the performance benefits of asynchronous replication with the consistency guarantees of synchronous replication. In this replication scheme the remote writes are performed asynchronously, allowing subsequent processing to proceed in parallel, however any externally visible event (such as an outgoing packet) must be blocked until all pending writes it depends on are committed both at primary and backup sites.

Pipelined Synchrony replication must guarantee a causal ordering between the externally visible events and the write requests they depend on. Since PipeCloud has no application visibility (it simply protects the disks of virtual machines), those dependencies are tracked by conservatively marking as dependent all writes issued before an outgoing packet (although some independent writes may be mistakenly marked, no dependent writes will ever be seen as independent).

All this virtualization-based approaches have the advantage of performing fast failover since they include a backup VM executing in a secondary site ready to take over when a disaster is detected in the primary infrastructure. On the other hand, they require virtual computing resources placed in a remote location, which implies high financial costs.

2.4 Filesystem Mirroring

Another common way of performing disaster recovery is by replicating data at the file system level. By continuously backing up the relevant files to remote storage facilities, a system is no longer susceptible to losing all its data if a catastrophic failure occurs in its primary infrastructure.

SnapMirror [16] is an asynchronous mirroring DR solution for network appliance file servers that make use of no-overwrite file systems (such as WAFL [26] and LFS [27]). This system periodically transfers and updates consistent file system snapshots to an online mirror capable of serving read only requests and becoming primary if a disaster occurs. It operates at the block level, and uses file system metadata to quickly identify the blocks that need to be synchronized with the remote mirror, leaving out data that was deleted or overwritten. By only transferring the relevant blocks, SnapMirror reduces the network bandwidth cost of the system and increases its performance. The benefits of this optimization are proportional with the update frequency, which is configured by the system administrators according with the performance, network bandwidth and data protection requirements of the system. SnapMirror also uses snapshots to make sure that the mirror remains in a consistent state, ready to come online, regardless of the moment when the primary site fails.

Seneca [17] is a robust asynchronous remote-mirroring protocol that provides resilience to disasters with low data loss. In this solution, a primary Seneca instance replicates its writes to a secondary Seneca instance placed at a remote location. The primary instance delays sending a batch of updates to the remote site, in order to perform write coalescing (reducing the volume of data to be propagated). It should be noted that this delay is limited in order to keep the copies as closely synchronized as possible. Afterwards, the writes are propagated to the secondary instance in an atomic way to avoid inconsistent states. This process of batching and coalescing write operations increases the overal performance of the system, and allows an efficient use of the WAN bandwidth between the primary and the secondary infrastructures. Finally, in order to tolerate crash faults, the authors propose that each Seneca instance include an active and a shadow node. The changes must be propagated to both nodes so that, when the active fails, the shadow can perform fail over.

The two file system mirroring solutions previously described have the advantage of allowing any application to protect its data, without requiring changes to its source code.

On the other hand, this solutions do not consider the semantics of the applications, which can result in inconsistent states after recovery. Additionally, these two solutions can only be deployed in infrastructures with computational power capable of executing their specific protocols (e.g., virtual machine instances in the cloud).

2.5 DBMS Recovery Solutions

Most of the Database Management Systems include features that can be used to perform disaster recovery. As PostgreSQL [18] is the DBMS studied in the scope of this project, we will now briefly present its recovery mechanisms [15], show how they can be used to tolerate disasters, and discuss their pros and cons. Such mechanisms will be explored in more detail in Section 3.2.3.

The first recovery mechanism is called *Continuous Archiving* and consists of performing a file-system-level backup of the database directory and setting a process (the archiver) that periodically backs up the completed WAL segments by executing a predefined shell command or script.[1]

This PostgreSQL functionality 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. One drawback of this approach is that the archiver process only operates over completed WAL segments, therefore it is not possible to have a tight control over the data that can be lost when a disaster occurs.

The other recovery feature of PostgreSQL is *Streaming Replication*. This feature consists of having a sender process in a primary server that streams the changes made to the database as they are generated to a receiver process hosted in one or several backup servers. These standby servers are kept synchronized with the primary by executing the received database changes. It is relevant to mention that, although this feature is asynchronous by default, it can be configured to function in a synchronous way.

This technique could also be used as a disaster recovery solution by placing one or more standby servers in a geographically distant site, for example in a virtual machine running in a cloud environment. However, this approach would have to include a remote computing instance capable of executing a PostgreSQL standby server (which may require a reasonable capacity). As a result, the monetary costs of this solution would be fairly high.

[1] *Write Ahead Log* (or *WAL*) [28] is a log that keeps records of every change made to the database. This log is stored in files called *WAL Segments*. When a WAL Segment is completed (i.e., it does not have room for more records) the following database changes are registered in another WAL segment. This will be covered in more detail in Chapter 3.

2.6 Cloud-Backed Storage Systems

In this section we will present solutions that rely on cloud storage services to store its data. Although the following solutions were not explicitly conceived to perform disaster recovery, we will discuss how they can be used to meet this goal.

Cumulus [29] is a system designed to perform efficient file system backups to cloud storage services over the internet. Cumulus makes very little assumptions about the remote service it uses (assumes only an interface with four basic operations over entire files). This aspect makes the system highly portable to any kind storage service.

The authors adopted a write-once model in which after a file is stored it can never be modified, only deleted. This model allows the clients to keep snapshots at different points in time and prevents snapshot corruptions due to failed backups. Furthermore, during recovery, this system supports both restoring entire snapshots and smaller selections of files.

Cumulus also issues a set of optimizations that include aggregating data from small files into larger files at the server (avoiding inefficiencies and lowering costs in the storage server and the network protocols) and dividing files into chunks (allowing it to store only the portions of a file changed since the previous snapshot). In short, Cumulus uses a specific data format that can be accessed in a very efficient way by its backup operations.

SCFS [30] is a file system that provides strong consistency and a near-POSIX semantics in a cloud-of-clouds environment. Besides the good durability grantees provided, this solution also allows its users to share files in a secure and fault tolerant way.

This file system is composed by the SCFS agents executing at the clients, the cloud storage services and a fault-tolerant coordination service. The coordination service is used to manage the metadata and to support synchronization (SCFS uses this service to build strong consistency on top of the eventually consistent cloud storage services).

The fact that this file system uses multiple clouds to store its data enables it to tolerate file corruptions and cloud unavailability [31]. In addition, this aspect eliminates the need of trusting any single cloud provider, avoiding problems such as lock-in. On the other hand, this decision increases the volume of data kept in the several clouds, which has negative consequences in terms of monetary cost. To mitigate this problem, the authors employ erasure coding techniques to reduce the size of the data stored.

SCFS also employs a set of optimizations in order to achieve high performance and scalability. These optimizations include storing the metadata of the files that are not shared in the cloud rather than in the coordination service, and implementing a local cache at the client. In terms of security, SCFS performs access control both at the cloud and the coordination service level. It should be noted that all the access control verifications are employed without trusting the SCFS agents executed in the client side. Additionally, all the data is encrypted before uploaded to the cloud, in order to achieve confidentiality.

In the work "*Building a Database on S3*" [32] the authors used a cloud storage service (specifically Amazon S3) as an underlying infrastructure to build a general-purpose database application. In this system the clients can: retrieve pages from S3, buffer them locally in memory or disk, update them, and write them back. All these remote 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 authors also devised a set of protocols that implement read and write operations at different levels of consistency and support a large number of concurrent clients (depending on the cloud capacity). The authors decided to sacrifice strict consistency and some ACID transaction properties (such as isolation) in order to achieve higher levels of scalability and availability. However, it should be noted that the protocols designed in this work offer some desired properties such as eventual consistency and atomicity.

All the previously described systems rely on cloud storage services to backup their data to remote locations. Although this approach is advantageous for being low-cost, it introduces some challenges, such as: all the logic must be implemented on the client side (because this type of cloud service can only be accessed through a narrow interface containing only basic operations); and in order to perform failover, it is necessary to have a computing facility that updates itself with the cloud storage.

Both *Cumulus* and *SCFS* can be used as a simple and reliable way to protect application data from disasters, by storing snapshots in a remote cloud infrastructure. This kind of approach has the disadvantages of not having control over the data that can be lost when a disaster occurs, and decreasing the performance of the application (since taking a consistent snapshot may require the system to stop operating for a short time). *Building a Database on S3* is a very interesting work that proves that it is possible to run a database system with loose consistency guarantees entirely on a cloud storage service. However, the performance of this kind of solution is orders of magnitude slower than existing production-level database management systems.

2.7 Discussion

In this chapter we covered a set of systems and approaches that are related to the work we performed in this thesis.

We stated by introducing the essential disaster recovery concepts, and by exploring how cloud services can be used to integrate diverse DR solutions suitable for different objectives.

Then we presented a series of solutions that can be used to perform disaster recovery. Most of these approaches require the existence of computing instances placed at the secondary infrastructure (e.g., a virtual machine running in the cloud). Although this aspect has the benefit of lowering the recovery time, it substantially increases the monetary and management costs.

In addition, the solutions described in this chapter were not specifically conceived to allow database management systems to recover from disasters. As a result, using such systems to meet this goal might bring severe consequences in terms of performance.

In this work we intend to develop a disaster recovery solution that introduces minimal monetary and management costs. This will be achieved by relying merely in cloud storage services to backup our data. Additionally, our solution will be specific to database management systems. This allows us to achieve better levels of performance, and to provide our clients with a tight control of the maximum amount of data that can be lost in a disaster.

Chapter 3

I/O in Database Systems

Databases are structured collections of interrelated data that are managed by software systems named Database Management Systems (DBMS). Database systems are widely used for storing and managing data in all kinds of applications.

In this chapter we will address how the input/output is performed in database systems and discuss the details of PostgreSQL [18]: the database management system in which our work will focus.

3.1 Implementation of Input/Output in Database Systems

DBMS typically deal with large amounts of information [33]. It is not possible to keep this amount of data completely in main memory, so the database management systems store its data in *data files* on disk and copy it to main memory as needed.

Disks are organized in *blocks* (also called *pages*), which are logical units of data that can be copied to and from main memory. A database element (such as a tuple or an object) is represented by a *record*, which is a set of consecutive bytes in a block (a block may contain multiple records). There are several types of records with different purposes (e.g., fixed length records, records with variable length fields, records with repeating fields, etc) that have different internal structures and thus must be handled differently.

Since the movement of data between disk and memory is very slow, the database systems use certain index structures to manage its data efficiently (for instance to minimize this data copies and to provide fast access to data items). The most common type of database index is the B-Tree [34] (represented in Figure 3.1). This index is a self-balancing tree data structure optimized for systems that read and write large blocks of data. The B-Trees are made of blocks with N keys and N+1 pointers to blocks in the level below, and only the leaf blocks point to the records.

The B-Trees are self-balanced and use blocks with many branches, this allows its height to grow slowly even for huge numbers of records. This is particularly interesting for systems like DBMS where some blocks of the tree (typically the lower ones) are stored

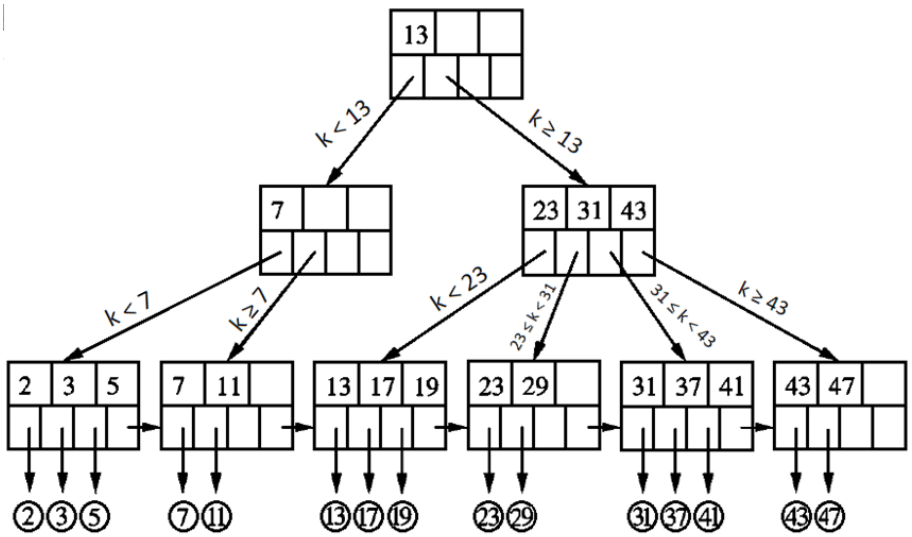

Figure 3.1: Example of a B-Tree (the circles and its numbers represent the records and its corresponding keys).

on disk, since it will allow the records to be found by accessing fewer blocks. For this reason, the B-Trees are excellent data structures for storing huge amounts of data for fast retrieval.

3.1.1 Failure Recovery in Database Systems

Computer systems fail for a variety of reasons. Consequently, database systems must protect its data from being lost in the occurrence of faults. The main technique for supporting data resilience is logging [28, 33].

Logging consists of safely storing all the database changes in a durable log. When a failure occurs, the log can be used to reconstruct the database during recovery. This technique is preferable over writing the changes directly to the data files because it performs sequential writes, increasing its performance. There are the following three different types of logging for database systems:

- **Undo Logging:** consists of writing to disk the log records that contain the changes to the database elements and its former values, then writing to disk the changed database elements themselves, and finally write to the log file the commit record. During recovery, the database state is restored by undoing all the uncommitted transactions. By requiring the data to be written to disk immediately after a transaction finishes, undo logging has the disadvantage of having a high I/O load.

- **Redo Logging:** in this form of logging the database elements can only be written to the data files after the changes to the database along with its new values and the commit record have reached the log files (this rule is called the write-ahead logging

rule). The recovery in redo logging involves redoing all the committed transactions. Although this logging scheme has a lower I/O load compared to the undo logging, it requires more memory per transaction since it has to buffer all the modified blocks until the transaction commits and the log records are flushed.

- **Undo/Redo Logging:** in this method the log records of the database changes, containing both its former and new values, are written to disk before the changes themselves (the commit record might reach disk before or after the database changes). The recovery is performed by redoing all the committed transactions and undoing all the uncommitted transactions. Undo/Redo Logging provides some flexibility regarding the order in which the writes are performed, but has the disadvantage of having larger log records.

Independently of the logging method used, the log files continuously record all the write operations issued on a database. This causes the log files to grow indefinitely, which is particularly undesirable because it will increase the amount of log records examined during recovery, thus increasing the recovery time. For this reason, database systems use a technique called checkpointing, in order to limit the length of log that must be examined during recovery.

Checkpointing consists of periodically making sure that all the operations present in the log are reflected in the data files. Consequently, at recovery time, the database system only needs to inspect the log records that follow the last performed checkpoint. In addition to reducing the recovery time, this technique allows the DBMS to reuse the disk space occupied by log records prior to the last checkpoint.

3.2 The PostgreSQL Case

PostgreSQL [18, 35] (or Postgres) is an open source object-relational database management system. Its extensibility and reliability are some of the reasons why this DBMS is widely used.

In this section, we provide an overview of PostgreSQL focusing on its interaction with the file system.

3.2.1 Architecture

PostgreSQL uses a client/server model. Its overall architecture [36–38] is represented in Figure 3.2.

The client side is composed by one application that connects remotely or locally to the server side through an API and issues commands to one or several databases. The client

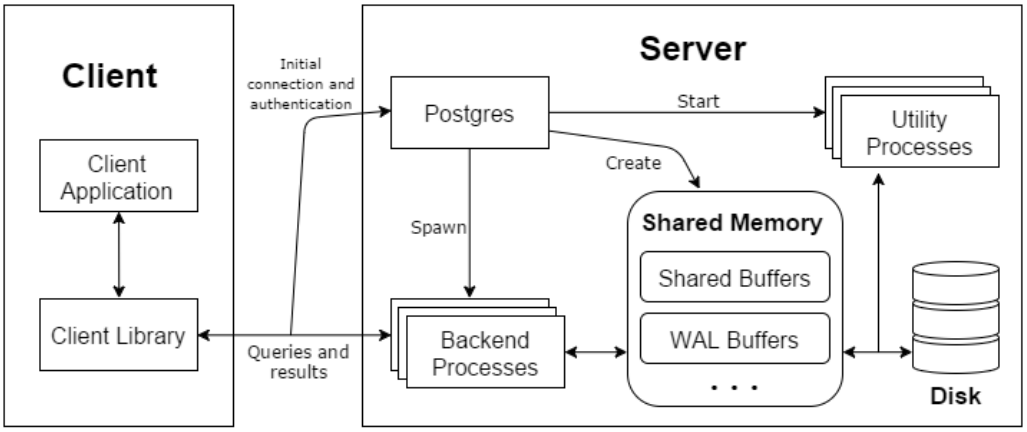

Figure 3.2: Architecture of PostgreSQL.

applications, also called frontends, can be very diverse (from command line administration tools to web servers that access databases) and are often developed by users as long as they follow the PostgreSQL protocol.

The server side is composed by a set of processes with different purposes, which access files and shared memory structures in order to provide a database service for the clients.

The main process in the server side is called `postgres` (or `postmaster`). Postgres is the first process that is started and it is always active. When launched, this process initializes the shared memory data structures, starts the utility processes (see Section 3.2.3) and then listens at a specified port for TCP or SSL client connections. When a client intends to connect to the server it firstly contacts the `postgres` process. Then, `postgres` performs authentication and, if the connection is valid, it spawns a new backend process to serve that client. From this point on, the backend is entirely responsible for that client session, thus the `postgres` process resumes listening for incoming client connections. When the client session is over, the backend process ends its execution.

PostgreSQL prefers this process per user model over a thread based solution mainly because it provides a greater level of isolation regarding memory access. Consequently, this design decision allows PostgreSQL to achieve higher levels of reliability and simplicity (since in the processes everything is private by default and the shared data structures are specifically defined by the programmers).

The `postgres` process is also responsible for other important functionalities such as performing recovery, managing the database files and initializing the shared memory structures.

3.2.2 Shared Memory

PostgreSQL uses shared memory to store database information that needs to be accessed by its processes [38]. This shared memory area includes many data structures with different purposes. We will only cover the shared memory areas that we consider relevant to our work, leaving out structures related with features such as vacuuming (see Section 3.2.3) or mutual exclusion (i.e., locking data items).

When a PostgreSQL instance is started, the `postgres` process allocates and initializes the shared memory data structures. Then, when spawning other processes, `postgres` ensures that they have access to those structures.

The biggest shared memory structure is the *shared buffers*. This memory region can be seen as an array of pages (by default 8kB long each) that store tuples from the database tables. When a backend server process intends to seek data, it begins by searching for the desired page into the shared buffers (there is a shared memory structure called buffer descriptors that tracks who is using each buffer and where they are located). If the desired page is not in memory, it is copied from the file system to the shared buffers. The backend process can then access the desired tuples in memory (each page has a structure containing pointers that allow us to quickly find the tuples we want). In PostgreSQL it is possible to create indexes such as B-Trees (recall section 3.1) that allow the backend processes to find the desired pages in a more efficient way.

As a consequence, all PostgreSQL I/O activity is performed through the shared memory. The fact that the backend processes can only access the database through the shared buffers ensures that they all have the same view of the database (the updates are made visible to all the processes).

Another important part of the shared memory are the *WAL buffers*.

PostgreSQL records all the changes made to the database in a redo log called Write Ahead Log (or WAL). Although the main purpose of the WAL is to perform recovery (i.e., restore the committed transactions that did not make it to the data files on the event of crash), the write ahead log has other applications such as point in time recovery (which is restoring a previous file system snapshot and executing WAL records until a specific point in time) and performing streaming replication (see Section 3.2.3).

When a client performs write operations to the database, those operations are not written to the data files immediately. Instead, they are performed in the shared buffers and records of those changes are written into the WAL buffers. The WAL buffers are flushed to permanent storage, specifically to the WAL segments, at every transaction commit (unless the asynchronous mode is enabled). The data files are later updated by the writer process or the checkpointer process (see next section). This is safe since all the write operations are in the WAL files and can thus be replayed during recovery in the event of crash.

The last shared memory structure we will cover here is the *CLOG buffers*. The Commit Log (or simply CLOG) holds the current status data of each transaction (a transaction

can be either in progress, committed, or aborted). When the CLOG buffers are filled, the least recently used buffer is flushed to permanent storage. Note that the CLOG only keeps the status of each transaction in progress, in opposition to the WAL, which registers all the transactions that were committed in the database.

3.2.3 Utility Processes

The utility processes are a set of processes that the `postgres` initiates when a PostgreSQL server is started [36–38]. These processes have different purposes and can be mandatory or optional. This section provides a brief description of each utility process.

Writer Process. The Writer process, also known as Background Writer (or BG Writer), is a mandatory process.

This process wakes up from time to time (the period is configurable), selects some specific dirty pages (i.e., pages in memory containing modifications that are not in disk yet) in the shared buffers, writes them to disk and removes them from the shared buffers. The algorithm that determines what pages to write uses information such as memory usage and which blocks were recently used. Afterwards, the BG Writer sleeps until its next timeout.

The Writer process ensures that free buffers are available for use and avoids spikes of I/O during the checkpoint (since it flushes dirty pages to disk between checkpoints).

WAL Writer Process. The WAL Writer is a mandatory process whose function is to periodically write and flush to disk the WAL records in the WAL buffers. By default, the WAL writer wakes up every 200 milliseconds to perform its activity, but this period is configurable.

Checkpointer Process. The Checkpointer is a mandatory process that is responsible for automatically performing checkpoints when a certain number of WAL segments is exceeded or when a timeout occurs (this two parameters are configurable, and their default values are respectively three segments and five minutes).

A checkpoint is a point in the transaction sequence at which all the data files are updated to reflect the information in the WAL. At checkpoint time, all the dirty pages in the shared memory are written and flushed to disk. The checkpointer process also marks those pages as clean and adds a special checkpoint record to the current WAL segment. In the event of a crash, the recovery module inspects the latest checkpoint record in order to determine the point in the WAL from which it must start to perform the redo operations.

It is important to mention some differences between the checkpointer and the writer process. First, the writer is responsible for writing specific dirty buffers to disk (based on an algorithm that considers the memory usage, which blocks were recently used, and

so fourth), whereas the checkpointer writes all the dirty buffers. Furthermore, the writer process only seeks dirty pages in the shared buffers while the checkpointer also inspects other areas of the shared memory such as the CLOG buffers.

Archiver Process. A possible strategy for backing up databases is to take a snapshot of the PostgreSQL file system and to set up a process that archives the WAL segments. In the event of recovery, the system can be brought to a consistent state by restoring the file system backup and executing the backed up WAL files.

The archiver is an optional process that is disabled by default. This process is responsible for capturing the data in the WAL files as soon as they are filled, and saving that data somewhere else before the files are recycled (i.e., renamed and reused as future WAL segments).

After activating this feature, the database administrator can define the archive timeout (which dictates how often the archiver checks for WAL segments ready to be archived) and specify the archive command. The latter is a shell command or script that will be executed by the archiver process in order to copy the segment files to the archive (this can be a tape drive, an NFS-mounted directory on another machine, and so on).

When WAL archiving is enabled, once a WAL segment is filled the WAL writer creates a file (named after the original segment file adding the suffix ".ready") in the `PGDATA/pg_xlog/archive_status` directory. When the archiver times out, it looks for ".ready" files and, if they exist, it executes the archive command parameterized with the name of the segment file (i.e., without the ".ready" suffix). Finally, the archiver renames the `<segment_filename>.ready` file to `<segment_filename>.done`.

Streaming Replication Processes. PostgreSQL includes a replication functionality for high availability called Streaming Replication. This functionality consists of having one primary server that serves clients and continuously replicates its WAL records to one or several standby servers. The standby servers can both reply to read-only requests and perform failover when the primary server fails.

The processes responsible for performing streaming replication are the WAL Sender and the WAL Receiver. In the primary server, the WAL Sender reads the WAL records as they are generated and streams them to the standby servers over TCP connections. In each standby server is running a WAL Receiver process that receives the WAL records from the primary server and executes them in order to remain synchronized.

Note that the sender process streams WAL records rather than sending WAL segments (like the archiver process does) because this way, the standby servers are kept more up-to-date. Furthermore, although PostgreSQL streaming replication is asynchronous by default (which means that when the primary server crashes some committed transactions may be lost), it is possible to configure this feature to be synchronous.

Autovacuum Launcher Process. The Autovaccum Laucher (also called Autovaccum Daemon) is an optional process which is enabled by default.

This process spawns worker processes (one can configure how many and how often those processes will perform its functions) whose function is to automate the execution of the VACCUM and ANALYSE commands. It uses the statistics generated by the Stats Collector process (described later) to decide which tables need to be vacuumed or analysed.

In PostgreSQL, when a UPDATE or DELETE command is executed, the tuples are not immediately removed from the data files. Instead, they are marked as deleted but the previous versions of those records remains in the data file so that the active transactions can see the data as it was before. When those tuples (often called dead tuples) become irrelevant, they should be marked as reusable so that the space they occupy in the data file can be used by subsequent write operations. This process of reclaiming the storage occupied by dead tuples by marking them as reusable is performed through the VACCUM command.

The ANALYSE command collects statistics related to the data distribution of the tables in the database. The number of distinct values or the most common values in one column are examples of statistics collected through this command. These statistics are used to help the query planner to determine more efficient execution plans for queries.

Stats Collector Process. The Stats Collector is an optional process (enabled by default) whose function is to collect information about the PostgreSQL server activity. The information this process collects is both permanently stored in cluster-wide tables and reported to other PostgreSQL processes (such as the backends and the autovacuum launcher).

The Stats Collector tracks the cluster activity and collects information such as the total number of rows in each table, how often the tables and indexes are accessed and data related to vacuum actions in each table. Although the activity of this process may introduce some additional overhead, it has advantages such as identifying objects that need to be vacuumed and provide systems administrators with information that can be helpful to configure the database server.

Logger Process. The Logger process (also called Logging Collector) is an optional utility process that is disabled by default. This process is responsible for logging the details of activity of the PostgreSQL instance.

All the other PostgreSQL processes (this includes the remaining utility processes, the backends and the `postgres` process) communicate with the Logger process in order to provide it with information about their activities. The Logger process writes the log messages it captures into log files according to its configuration.

Directory	Description
PG_VERSION	Text file containing the version number of PostgreSQL.
base	Directory that contains the database files.
global	Directory that contains tables that keep track of the cluster.
pg_clog	Directory that contains the CLOG files.
pg_multixact	Directory that contains multitransaction status data.
pg_notify	Directory that contains LISTEN/NOTIFY status data.
pg_serial	Directory that contains information about committed serializable transactions.
pg_snapshot	Directory that contains exported snapshots
pg_stat_tmp	Directory that contains temporary files used by the stats collector process to communicate with the other processes.
pg_subtrans	Directory that contains subtransaction status data.
pg_tblspc	Directory that contains symbolic links to tablespaces.
pg_twophase	Directory that contains state files for prepared transactions.
pg_xlog	Directory that contains the WAL segments.
postmaster.opt	Text file containing the command line options the server was last started with.
postmaster.pid	Lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (empty on Windows), first valid listen_address (IP address or *, or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown).

Table 3.1: Contents of PGDATA (Adapted from PostgreSQL Documentation: "Database File Layout" [38]).

3.2.4 Table-File Mapping

All the data that a PostgreSQL server manages is located in a directory known as PGDATA (one machine can host several servers and thus have several PGDATA directories) [38]. Table 3.1 presents all the files and directories contained in PGDATA, as well as a short description of their purpose.

Let us now introduce briefly how the file system is used to store databases. Figure 3.3 represents the relation between the PostgreSQL databases and the file system (we have not included all the PGDATA subdirectories and files for a sake of simplicity).

At the top of this figure we can see two databases, the left one is zoomed in so that we can see its relation to the files in the base subdirectory. The object identifier (OID) of database 1 is 16386 so its data will be written in the directory `PGDATA/base/16386`.

As we can see, each table stores its data in a file named after its OID (e.g., the OID of Table 1 is 16428). In addition, each table may have a free space map (stored in the `<OID>_fsm file`) which is a binary tree that tracks the unused space inside the table,

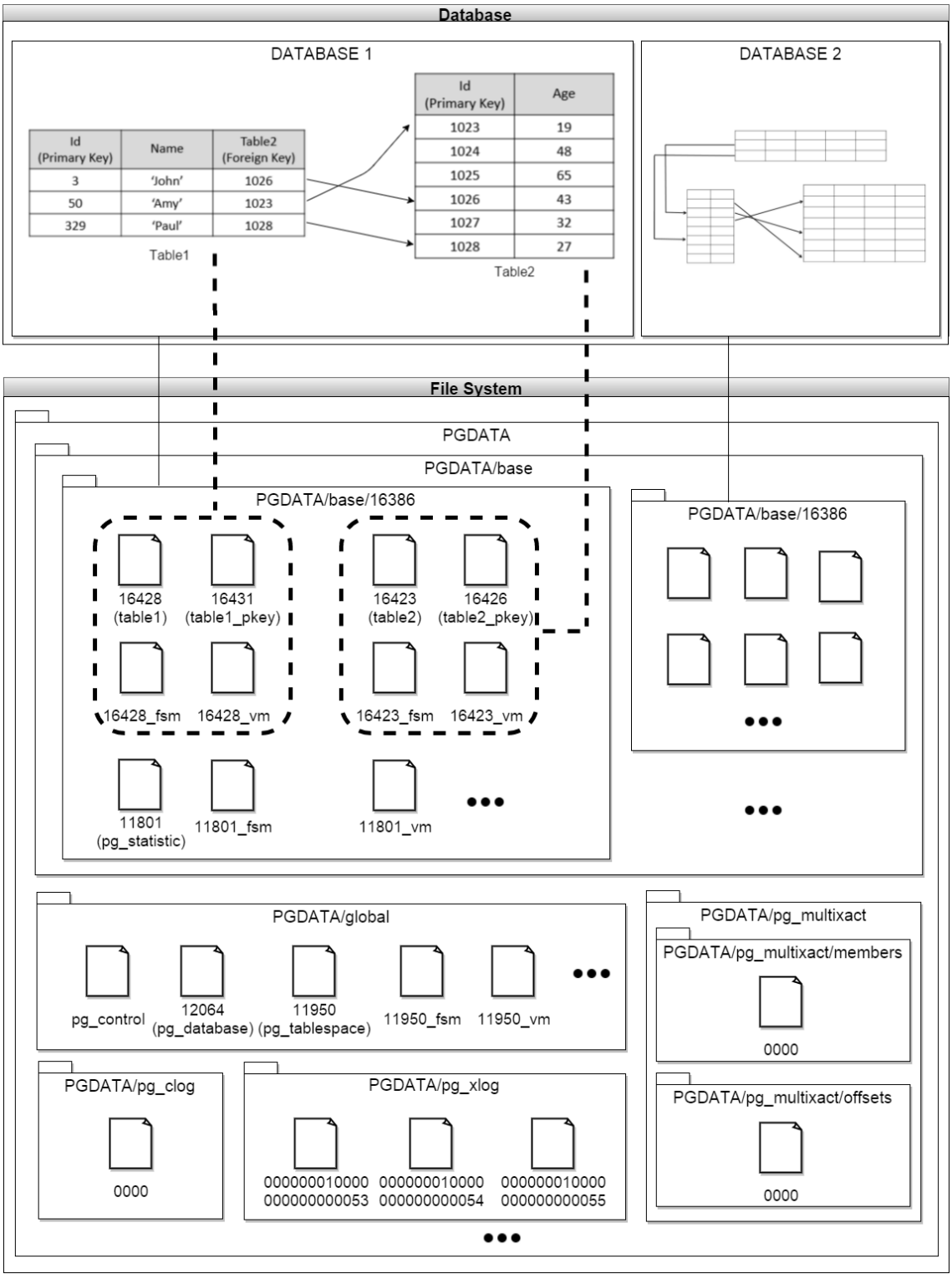

Figure 3.3: Relation between the PostgreSQL databases and the file system.

a visibility map (stored in the <OID>_vm file) which is a bitmap that keep track of the pages which have only tuples that are visible to all active transactions (and therefore do not need to be vacuumed), and other indexes such as the primary key index (stored in a file named after its own OID).

In the PGDATA/global directory we can observe some examples of cluster-wide tables and note that they also may have its vm and fsm files.

PGDATA/pg_xlog is one of the most important directories in PostgreSQL. As described before, PostgreSQL continuously produces WAL records and stores them into WAL segments (which, by default, are 16MB files that contain blocks of 8kB) within this directory. These files have numeric names that reflect their position in the WAL sequence. PostgreSQL switches to the next segment file when the current one is filled up or when it is forced to do so (e.g., when a system administrator executes the pg_switch_xlog function). PostgreSQL keeps a set of WAL segments and recycles them when they are no longer needed (e.g., the WAL segments that precede a checkpoint) by renaming and reusing them.

3.2.5 Input/Output Operations

Understanding exactly how PostgreSQL performs its I/O operations in the file system is crucial to devise an efficient disaster recovery solution for this system. In this subsection, we will expose this subject regarding some of the most common database functionalities. The following data was collected through direct observation (using observability tools such as strace [39] and lsof [40]) as well as through the reading of documentation and the source code of PostgreSQL.

Table 3.2 shows the relation between the database operations and the invoked system calls (the most relevant calls are represented in bold). To simplify the presentation, we have not represented the open() calls, since they are issued whenever is necessary to access a closed file. In this table we have represented the current WAL segment as WAL, the current CLOG file as "CLOG", the data file of the table on which we are operating as table_name, and the primary key index files of the table_name as table_pkey.

Database Operations	Disk Operations

CREATE TABLE	`for each file in [table_file, table_pkey] {`
	`  lseek(file, 0, SEEK_END)`
	`}`
	`write(table_pkey, 8192);lseek(table_pkey, SEEK_END)`
	`fsync(table_pkey)`
	
	`lseek(table_file,SEEK_END);lseek(table_pkey,SEEK_END)`
	`close(table_file); close(table_pkey)`
	
	`lseek(WAL, <lastPage>, SEEK_SET)`
	`write(WAL, 8192)`
	`fdatasync(WAL)`

INSERT	`lseek(WAL, <lastPage>, SEEK_SET)`
	`write(WAL, 8192)`
	`fdatasync(WAL)`

DELETE and UPDATE	`lseek(table_file, 0, SEEK_END)`
	`lseek(table_pkey, 0, SEEK_END)`
	
	`lseek(WAL, <lastPage>, SEEK_SET)`
	`write(WAL, 8192)`
	`fdatasync(WAL)`

SELECT (with the tables in memory)	`lseek(table_file, 0, SEEK_END)` `lseek(table_pkey, 0, SEEK_END)`

SELECT (without the tables in memory)	`for each file in [pg_namespace,`
	`  pg_namespace_nspname_index,`
	`  (...), table_pkey, table_file] {`
	`  for i in nTimes {`
	`  lseek(file, <desiredPage>, SEEK_SET)`
	`  read(file, 8192)`
	`  }`
	`}`
	`lseek(table_file, 0, SEEK_END)`
	`lseek(table_pkey, 0, SEEK_END)`

INSERT, DELETE and UPDATE (inside a transaction)	`lseek(table_file, 0, SEEK_END)`
	`lseek(table_pkey, 0, SEEK_END)`
	`lseek(table_file, 0, SEEK_END)`

COMMIT	`lseek(WAL, <lastPage>, SEEK_SET)`
	`write(WAL, 8192)`
	`fdatasync(WAL)`

Database Operations	Disk Operations

Checkpoints	for each file in [CLOG, pg_subtrans/0003,
	pg_multixact/offsets/0000, table_file,
	table_pkey, table_vm, table_fsm] {
	for page in [dirtyPages] {
	lseek(file, page, SEEK_SET)
	write(file, 8192)
	}
	fsync(file)
	close(file)
	}
	lseek(WAL, <lastPage>, SEEK_SET)
	write(WAL, 8192)
	fdatasync(WAL)
	close(WAL)
	
	write(pg_control, 232)
	fsync(pg_control)
	close(pg_control)
	
	for each x in [WAL segments] {
	stat(x)
	}
	
	for each file in [WAL, subtrans,
	pg_multixact/offsets/0000,
	pg_multixact/members/0000]{
	openat(AT_FDCWD, file, O_RDONLY|O_NONBLOCK|
	O_DIRECTORY|O_CLOEXEC)
	getdents(file, /* 5 entries */, 32768)
	getdents(file, /* 0 entries */, 32768)
	close(file)
	}

Table 3.2: System calls performed by each database operation (assuming that the tables are already in memory unless otherwise specified).

Note that, as discussed previously, PostgreSQL performs most of its read and write operations in pages of 8kB. Let us now present a brief explanation of this operations.

`CREATE TABLE.` When creating a table, PostgreSQL begins by creating the files in which the table will be stored. In order to determine the filenames for this table, this step may require opening and reading several files in the `PGDATA/base` directory, if such information is not in memory yet.

Then the backend process performs the required initialization writes in the files it just created (for instance if the table has a primary key column, it writes the pkey B-tree structure to the pkey file) and closes them.

Lastly, a write operation in the WAL file is issued and flushed to disk.

`DELETE`, `INSERT` and `UPDATE`. These three operations exhibit very similar behaviours.

First, the backend process rewrites the last page of the WAL including the operation performed (`DELETE`, `INSERT` or `UPDATE`) and its arguments (if the current WAL segment is full, this write operation is performed in the first page of a new WAL segment). Afterwards, the operation is executed in memory and a response is sent to the client.

`SELECT`. When the table containing the data we are willing to retrieve is in memory, this operation performs no read operations at all. If, on the other hand, the desired data is not in memory yet, the backend process reads the files related to the tables accessed by the query (it may also be necessary to read some specific PostgreSQL files in order to determine which files store the tables and indexes we are looking for).

Transactions. A transaction is a sequence of database operations (typically SQL commands) whose results (either all or nothing) will be made visible only when the transaction commits. In PostgreSQL a transaction can be issued through the commands `BEGIN` and `COMMIT`.

When executing `BEGIN` commands, the backend process do not perform any I/O operation whatsoever. For that reason, we have not included this operation in Table 3.2.

Inside a transaction, we have observed the operations: `INSERT`, `UPDATE`, `DELETE` and `SELECT`. When an `INSERT` command is executed, the backend process does not issue any relevant system call. The remaining three operations observed led to the execution of `lseek`'s over the `table_file` and `table_pkey` files.

When the command `COMMIT` is executed, the backend process writes to the current WAL file and flushes it to the storage device. This is the only write operation that we have observed during transactions.

We can notice that a transaction leads to the same writes as an operation like `UPDATE`. The only difference is that a transaction batches the writes of several operations and executes them once, after the transaction is committed.

Checkpoints. In the operations seen so far no write is issued in the files that store the tables. That is because those files are updated during checkpoints (or when the writer process wakes up). Recall from Section 3.2.3 that the checkpointer process is responsible for flushing the dirty pages from the shared memory to disk from time to time (or when the number of WAL segments exceed a certain threshold).

The checkpointer starts by writing and flushing the dirty buffers to the appropriate files. Notice that the files to be written depend on the database operations that were performed since the last checkpoint, so some of the data in Table 3.2 is relative to our specific executions. Then, the checkpointer writes one page in the WAL file (containing the checkpoint record) and a few bytes in the `pg_control` file (containing the checkpoint's position).

3.3 Final Considerations

In this chapter we have covered how Input/Output operations are performed in database systems, focusing on the database management system our work will use. The subjects discussed in this chapter will dictate the decisions we will take when designing our disaster recovery system.

Chapter 4

GINJA: A Low-cost Database Disaster Recovery Solution

In this chapter we describe GINJA: a low-cost disaster recovery solution for database management systems that provides fine-grained control over the data that can be lost when a disaster strikes, while introducing a minimal performance overhead to the DBMS.

In Section 4.1 we state the principles and assumptions of GINJA. In Section 4.2 we present our general architecture. Afterwards, in Section 4.3 we introduce the usage and pricing model of the cloud storage services. Section 4.4 presents our configuration parameters and explore how they allow a tight control over the maximum data loss caused by disasters. In Section 4.5 we explain the data model used to store information in the cloud and argue that it provides the right balance between cost and efficiency. Then, in Section 4.6 we explain in detail the algorithms that dictate how GINJA operates (specifically how it affects the DBMS workflow and performs cloud synchronizations). Finally, in the last section we end up with a few considerations that summarize this chapter.

4.1 Principles and Assumptions

The most essential design principle behind GINJA is minimizing monetary costs. Consequently, we used one of the cheapest services available to build our secondary infrastructure, and used this service in the most cost-efficient way possible (taking into account its pricing model).

Another fundamental factor that influenced GINJA was providing a fine-grained control over the data that can be lost due to a disaster. This resulted in the creation of a flexible configuration model that allows the users to take decisions in terms of durability, performance and monetary cost.

Additionally, GINJA was designed to be as portable as possible. As a result, we did not perform modifications to the DBMS, and created a modular solution that can be easily extended to support other database management systems.

Finally, it must be mentioned that GINJA is a DR solution for transactional database management systems. Specifically, GINJA assumes that the DBMSs it protects use a *Write-Ahead Log* [28] to register its database updates, and perform checkpoints periodically (recall Chapter 3). Examples of such database systems are PostgreSQL [18] and MySQL using the InnoDB Storage Engine [41].

4.2 Architecture

The general architecture of GINJA is represented in Figure 4.1. Our solution is a specialized file system in user space that intercepts the operations called over the files of the DBMS and backs the relevant data up to a cloud storage service in a cost-efficient manner.

An alternative solution would consist of modifying the DBMS to include our DR strategies. This approach has the disadvantage of being less portable, as we would have to alter the code of all the database management systems we intend to support (and this process would be repeated every time a new version was released). Furthermore, it would be impossible to use GINJA to protect proprietary database systems from disasters.

Another possible approach would pass by employing our algorithms at the block device level just like other DR systems [16, 17]. This abstraction level allows us to handle indistinct data blocks, without providing us with any application visibility. Consequently, as one of our goals is to support a fine grained control over the database operations that can be lost due to a disaster, we chose not to adopt this approach.

For the previous reasons, we believe that the file system is the right level of abstraction to implement our disaster recovery solution. First, implementing GINJA at this level is advantageous because it eases its integration with the DBMS: the administrator only needs to mount the file system in the database directory and perform some configurations. In addition, this design decision allows our system to be easily extended to provide its service to other DBMSs such as MySQL [19] or Oracle [42]. On the other hand, implementing our DR solution as a file system introduces the challenge of identifying the operations that are being performed on the database just from the file system calls that are captured. To cope with that, we used the information covered in Sections 3.2.4 and 3.2.5 to devise a set of algorithms that define the actions to perform when certain events occur (such as writing to a WAL segment and performing a checkpoint).

GINJA relies on cloud storage services (e.g., Amazon S3, Azure Blob Storage) to store its data in a remote site. We choose such services as a secondary infrastructure because they have the potential of lowering both our monetary and our management costs.[1]

This decision has a great impact on the design of our solution. First, storage clouds provide narrow REST interfaces containing only a few basic operations. As a conse-

[1]Besides being cheaper, the cloud storage services are substantially easier to manage than computing cloud instances, which require setting up a firewall, a public IP address, and so forth.

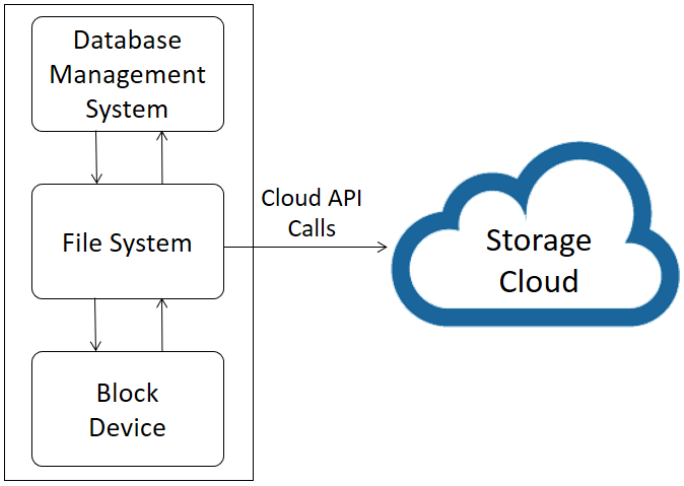

Figure 4.1: General architecture of GINJA.

quence, we have to implement all the logic of our system in the client side using only the available cloud operations. 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 should also be noted that building our secondary infrastructure entirely on top of a service with no computational power can brings consequences in terms of recovery time. This is because in order to perform failover it is necessary to install GINJA in a computational instance, and execute it in recovery mode to download all the data present the cloud. This procedure can be quite expensive in terms of time, specially when there is a large amount of data to be downloaded. Fortunately, this can be solved by performing failover using a virtual machine placed in the cloud infrastructure that stores GINJA's data, which substantially reduces the download time.

4.3 Using Cloud Storage Services

Cloud Storage Services provide a virtually infinite storage facility that can be accessed remotely through a REST interface. In this paradigm, data is stored in binary objects associated with keys. The keys are used to access the objects present in the cloud, thus each key can be associated with at most one object.

The fundamental operations supported by this kind of service are:

- *PUT(key, object)* – Upload a cloud object associated with a given key to the cloud;
- *GET(key)* – Download the object associated with the given key from the cloud;

- *DELETE(key)* – Delete the object associated with the given key from the cloud;
- *LIST()* – List the available objects stored in the cloud.

The pricing model of these services takes into account the volume of data stored in the cloud, the number of operations executed and the amount of data downloaded from the service (the upload traffic is free as an incentive to put more data in the cloud). Different cloud operations have different prices: *PUT* and *LIST* are the most expensive operations, *GET* is fairly cheaper, and *DELETE* is free.

During normal operation, GINJA will only execute *PUT*s. As this is one of the most expensive operations, we must batch our cloud writes as much as possible before performing cloud synchronizations.

The *GET* operation will only be used during recovery. This means that the overall cost of our solution is higher when it is necessary to perform recovery (which is advantageous because disasters are rare events). In that situation, GINJA will download the relevant objects from in the cloud, which brings costs relative to the number of *GET* operations executed and the volume of data transferred in those operations.

As the volume of data kept in the cloud influences the price of our solution, and there is no cost associated to the execution of *DELETE*s, GINJA will leverage this operation to reduce the amount data maintained in the cloud. Of course this removal has to be done in a safe manner (i.e., only cloud objects that are no longer relevant can be deleted), so that the DBMS files can be consistently reconstructed during recovery.

4.4 System Parameters

We deal with the trade-off between performance and data protection [6] by allowing the users to decide the maximum amount of data that can be lost when a disaster occurs. Thus, instead of following a completely synchronous or asynchronous approach, we have defined a model that allows our users to define the desired level of synchrony. Furthermore, as sending data to the cloud has its costs, our model also delegates to our users the performance vs cost trade-off. This model includes the following concepts:

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

Batch dictates how the DBMS data is backed up to the cloud, whereas *Safety* defines the durability guarantees provided by our solution, i.e., the RPO of the DBMS.

These variables can be defined both in terms of time and number of database updates, which results in the four configuration parameters presented in Table 4.1. This aspect allows users to have a tighter control over the behaviour of our system. The variables *B*

Parameter	Description
B	The maximum number of database updates that can be included in each cloud synchronization.
T_B	The maximum amount of *time* to start a cloud synchronization for a non-empty set of database updates.
S	The maximum number of database updates that can be lost in the event of a disaster.
T_S	The maximum amount of *time* in which the updates can be lost in the event of a disaster.

Table 4.1: GINJA's configuration parameters.

and S define a threshold of database updates that trigger GINJA to perform its actions, even when the database system receives bursts of write requests. For situations in which the DBMS receives a low rate of write requests, the variables T_B and T_S establish a maximum amount of time in which our solution must replicate data to the cloud and block the DBMS, respectively. More specifically, each cloud synchronization includes B database updates, or the (non-empty) set of database updates executed in the last T_B seconds. Likewise, GINJA blocks the DBMS when more than S updates are executed but not uploaded to the cloud or more than T_S seconds have passed since the last successful cloud synchronization. For a sake of simplicity, from this point forward we will refer mostly to the B and S parameters, as they are the ones that matter when the DBMS is under a significant load.

Figure 4.2 illustrates an example of how the parameters B and S influence the execution of GINJA. 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, our system blocks the DBMS until a positive acknowledgement is received from the pending cloud synchronizations.

Ideally, B should be substantially lower than S, so that GINJA does not block or interfere with the DBMS performance during regular operation. If some adverse conditions occur (such as network instability or cloud errors), then the S parameter ensures that no more than a certain threshold of data is ever lost due to a disaster.

To summarize, GINJA's configuration model allows its users to perform a proper configuration according to their specific application requirements. S defines the degree of data protection that our system provides, which as a negative impact in the performance of the DBMS. B allows our users to control how the cloud synchronizations are performed, which influences the monetary cost of the system and smooths the performance limitations introduced by the S parameter.

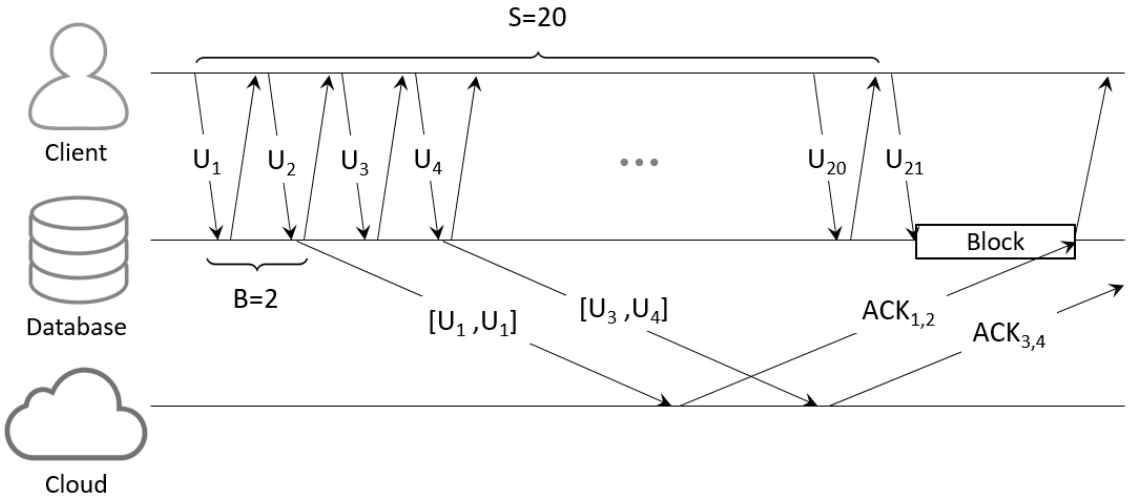

Figure 4.2: Influence of *B* and *S* in the execution of GINJA. In this example *B*=2, thus each cloud backup includes two database updates. It is possible to observe that GINJA blocks the DBMS whenever more than 20 database updates (which is the value of *S*) are executed without being acknowledged by the cloud.

4.5 Data Model

As we have previously mentioned, one of the drawbacks of using storage clouds is the narrow interface they provide. Specifically, one of the main limitations introduced by using this type of service is the fact that it does not support updating parts of existing objects in the cloud.

As a consequence, we have developed a specialized data model that allows our DR solution to synchronize file updates as they are issued locally, and reconstruct those files from the objects present in the cloud when necessary. This model is optimized to reduce the total volume of data kept in the cloud and to minimize the number of cloud operations executed (as these are the only factors that influence GINJA's monetary cost in the absence of disasters).

Our data model includes the following two types of cloud objects:

- *WAL Objects* – which contain chunks of information present in the local WAL segments. This means that the content of each local WAL segment is stored in several cloud objects. The WAL objects are named following the format `WAL/<timestamp>_<filename>_<offset>`, in which *timestamp* is a sequential number that establishes total order on the WAL objects, *filename* is the name of the corresponding WAL segment (i.e., the name of the log file segment), and *offset* is the position of its content in the WAL segment. This way, initially the whole content of a local WAL segment is stored in a cloud object called `WAL/0_<filename>_0` and, as writes are made to that file, new objects (such as `WAL/1_<filename>_8192`, `WAL/1_<filename>_16384`, etc) are uploaded to the cloud.

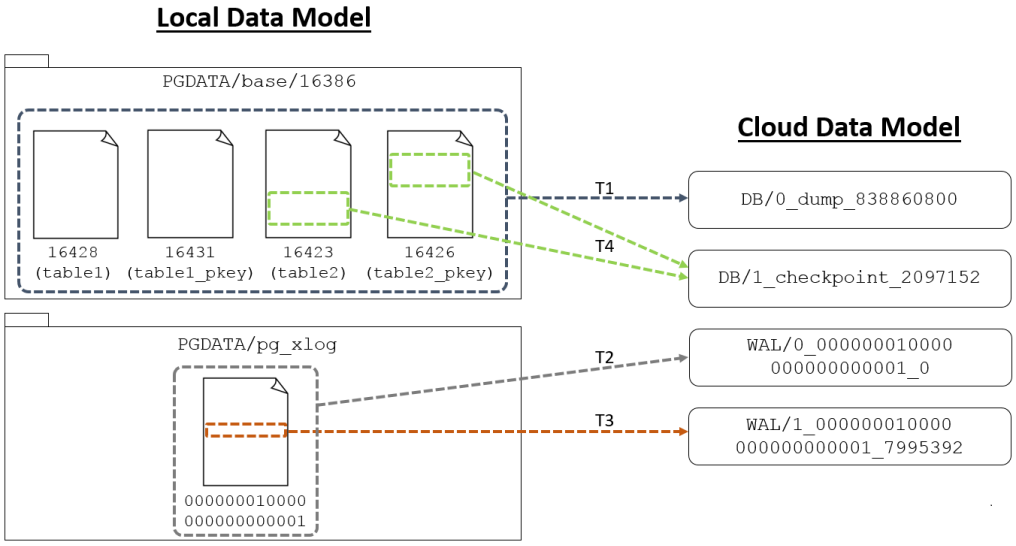

Figure 4.3: Cloud data model.

- *DB Objects* – which store information relative to all the relevant database files excluding the WAL segments. There are two types of DB objects: *dumps* and *incremental checkpoints*. The dumps store a snapshot of the state of the local database files in a certain moment. The checkpoint DB objects store incremental updates relative to the previous dump object present in the cloud. The DB objects are named following the format `DB/<timestamp>_<type>_<size>` that contains its *timestamp*, its *type* (`"dump"` or `"checkpoint"`), and its *size*. The timestamp of a DB object is the timestamp of the last WAL object uploaded to the cloud before the beginning of the checkpoint that originated this DB object. This means that after a DB object with the timestamp *ts* is successfully uploaded, all the WAL objects with a timestamps lesser than *ts* can be safely deleted from the cloud, as they are no longer relevant (i.e., their WAL data will not be used during recovery).

  It is important to mention that we limit the maximum volume of each DB cloud object to 1GB (e.g., a dump of a 10GB database results in 10 DB cloud objects).[2]

Note that while the DB objects store data from several local files, each WAL object store data relative to only one WAL segment. This is because, as covered in Chapter 3, the WAL contains records of all the database updates executed in the DBMS, which can be used to recover from failures. Hence, we use the write operations performed over the WAL segments to backup the database updates according to the parameters described in the previous section.

Figure 4.3 presents an example of the objects that can be in the cloud during an execution of GINJA. Initially the entire content of the database files was uploaded to the cloud in

[2]This parameter can be configured with other values.

the form of a 800MB dump DB object (instant T1). In addition, the whole content of the current WAL segment was uploaded as a WAL cloud object (instant T2). Then, *B* database updates were executed, which resulted in the second WAL object visible in our example (instant T3). At last, the database management system executed some write operations to the database files during a checkpoint, triggering GINJA to generate the checkpoint DB object `DB/1_checkpoint_2097152`, and upload it to the cloud (instant T4).

Finally, it is relevant to mention that GINJA supports compressing and encrypting all the data sent to the cloud. *Compression* reduces the volume of data uploaded, which reduces the synchronization latencies as a consequence. *Encryption* can be employed to ensure the confidentiality of the database information stored in the cloud infrastructure.

4.6 Algorithms

As discussed in Chapter 2, data replication over wide area networks is not an easy task, as it affects aspects such as performance, cost and reliability. In this section we describe the algorithms used by GINJA to cope with such challenges in a safe and efficient manner.

All the algorithms use a data structure named *cloudView* to keep track of the existing objects in the cloud. This structure is very important to upload, delete and download cloud objects. *cloudView* is initialized when the system is started and updated every time a cloud operation is performed. This way, we make sure that *cloudView* remains updated without having to continuously execute *LIST* operations to the cloud.

Initialization. Algorithm 1 describes the steps performed by GINJA during initialization.

First, GINJA initializes all the data structures, threads and variables necessary to the system (Lines 1-6). Afterwards, GINJA performs the actions relative to the configured initialization mode. The available modes are: *Init*, *Reboot* and *Recovery*.

The *Init* mode is used for the system to create a cloud backup of an existing database. In this case all the relevant database files are uploaded to the cloud before the file system is mounted and the DBMS starts operating. This type of initialization results in one WAL object for each local WAL segment (Lines 7-11) and one dump DB object (Lines 12-16).

The *Reboot* mode is meant to be used when a system administrator performs a safe reboot of the system. This mode assumes that the data in the cloud is synchronized with the local files of the database. Hence, when GINJA is initialized in this mode, it simply executes a *LIST* operation to discover the objects present in the cloud and updates the *cloudView* data structure with that information (Lines 17-19). For only requiring the execution of one *LIST* operation, this is the fastest initialization mode of the three.

The *Recovery* mode is used to rebuild the database files from the objects in the cloud. In this mode, the system downloads all the relevant data from the cloud, and reconstructs the DBMS's files as they were after the last successful synchronization. During *Recovery*, GINJA starts by discovering the objects present in the cloud and updates the *cloudView*

data structure with such information (Lines 20-22). Then, it downloads the most recent dump DB object in the cloud, and writes its data in the corresponding local files (Lines 23-25). Afterwards, the local database files are updated with the incremental checkpoint objects in the cloud that follow the dump object downloaded (Lines 26-32). Finally, GINJA obtains the WAL data written after the last checkpoint from the WAL objects present in the cloud (Lines 33-36).

Database Commits. Algorithm 2 describes how GINJA reacts to committed database operations. In this algorithm we ensure that the parameters described in Section 4.4 are respected.

As can be seen in the algorithm, when a commit is written to the WAL, we add this write to a queue for processing in background and check if the S and T_S parameters are not violated (Lines 4-6). If the commit queue increases too much (i.e., the uploads take too long), the S or the T_S parameters will be reached and GINJA will block the DBMS until the pending database updates are safely replicated in the cloud.

The writes are consumed from the queue asynchronously and uploaded to the cloud. First, the writes are consumed respecting the B and T_B parameters (Lines 8-9). The updates read from the *commitQueue* are aggregated into only one for each WAL segment (Line 10), and written to the cloud (Lines 11-14).[3] This is really important because most databases write pages in the log, and many times 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's executed in the cloud and, as a result, the monetary cost of our DR solution decreases.

Note that it is possible to have several threads uploading cloud objects in parallel, which brings great benefits in terms of performance (see Chapter 6). On the other hand, it is no longer guaranteed that the WAL cloud objects are uploaded by *ts* order. This introduces serious consequences that must be taken into account.

First, in the worst case scenario, a disaster may occur at a moment when the most recent WAL updates are 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 cloud objects that have consecutive timestamps (see Line 34 of Algorithm 1). Consequently, to guarantee that the maximum amount of updates lost in case of disaster respect the S and T_S parameters, GINJA unblocks the DBMS only after uploading all WAL objects with consecutive *ts* numbers. 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 only reset to unlock the DBMS if and only if the WAL object previously uploaded can be used to recover from a disaster that would occur immediately (Lines 17-20).

[3] Note that the WAL segments are typically large files (e.g., 16MB in PostgreSQL, 48MB in MySQL). Consequently, this aggregation results normally in only one cloud object.

Algorithm 1 Initialization.

Initialization:

```
1: cloudView ← ∅                                          ▷ Used in all Algorithms
2: TaskT_B.startTimer(T_B)                                ▷ Used in Algorithm 2
3: TaskT_S.startTimer(T_S)                                ▷ Used in Algorithm 2
4: for i=1 to nThreads do                                 ▷ nThreads is configurable
5:     runInBackground(CommitThread)                      ▷ Used in Algorithm 2
6: runInBackground(CheckpointThread)                      ▷ Used in Algorithm 3
```

Init:

```
7: currentTs ← 0
8: for each file in Local WAL Segments, in increasing order do
9:     cloud.PUT("WAL/"+currentTs+"_"+file.name+"_0", file.content)
10:    cloudView.addWAL(currentTs, file.name, 0)
11:    currentTs ← currentTs + 1
12: dbObject ← ∅
13: for each file in Local DB Files do
14:    dbObject.add(file.name, file.content)
15: cloud.PUT("DB/0_dump_"+dbObject.size, dbObject)
16: cloudView.addDB(0, "dump", dbObject.size)
```

Reboot:

```
17: cloudObjects ← cloud.LIST()
18: for each obj in cloudObjects do
19:    cloudView.add(obj)
```

Recovery:

```
20: cloudList ← cloud.LIST()
21: for each object in cloudList do
22:    cloudView.add(object)
23: dump ← cloud.GET(mostRecentDump(cloudList.dbObjects))
24: for each file in dump do
25:    writeLocally(file.name, 0, file.content)
26: checkpoints ← newerThan(cloudList.dbObjects, dump.ts)
27: maxCkptTs ← dump.ts
28: for each obj in checkpoints, in increasing ts order do
29:    currentCkpt ← cloud.GET(obj)
30:    for each file in currentCkpt do
31:        writeLocally(file.name, file.offset, file.content)
32:    maxCkptTs ← obj.ts
33: segments ← newerThan(cloudList.walObjects, maxCkptTs)
34: for each obj in segments, in increasing ts order and with no gaps do
35:    content ← cloud.GET(obj)
36:    writeLocally(obj.filename, obj.offset, obj.content)
```

Algorithm 2 Database Commits.

Variables:

commitQueue ← ∅ ▷ Holds all the pending synchronizations
timeoutT$_S$ ← false
timeoutT$_B$ ← false

Upon write(WAL_segment, offset, content) **do:**

writeLocally(WAL_segment, offset, content)
commitQueue.put(⟨WAL_segment, offset, content⟩)
wait until commitQueue.size ≤ S **and** timeoutT$_S$ = false

CommitThread:

loop
    wait until commitQueue.size ≥ B **or** timeoutT$_B$ = true
    updates ← commitQueue.getNextBatch() ▷ Does not remove from the queue
    aggUpdates ← aggregateUpdates(updates)
    for u **in** aggUpdates **do**
        ts ← cloudView.getNextWALts()
        cloud.PUT("WAL/"+ts+"_"+u.filename+"_"+u.offset, u.content)
        cloudView.addWAL(ts, u.filename, u.offset)
    TaskT$_B$.resetTimer()
    timeoutT$_B$ ← false
    wait until commitQueue.lastBatchElements() = updates
    commitQueue.removeLastNElements(updates.size) ▷ Removes from the queue
    TaskT$_S$.resetTimer()
    timeoutT$_S$ ← false

TaskT$_B$ (upon timeout):

if commitQueue.size > 0 **then**
    timeoutT$_B$ ← true ▷ Trigger the commitThread to start uploading

TaskT$_S$ (upon timeout):

if commitQueue.size > 0 **then**
    timeoutT$_S$ ← true ▷ Block the DBMS

Checkpoints and Garbage Collection. Our model assumes that the database management system performs periodic checkpoints that consist of writing dirty pages in memory to the database files and marking the WAL as applied up to that point (recall Chapter 3).

Algorithm 3 describes how GINJA handles checkpoints. As performance is one of our key concerns, we decouple the local checkpoints with the cloud synchronization of the database files (Lines 5, 19-20). As all the database updates are written to the WAL segments and synchronized as described in Algorithm 2, no database update will ever be lost due to this decision. However, we must be cautious when dealing with the checkpoint records (marking a checkpoint completion) in the WAL segments as these records mark the starting point for replaying the log after a failure.

When a checkpoint is finished, PostgreSQL writes a checkpoint record to the current WAL segment and then writes the position of that WAL record in the `pg_control` file (this can be observed in Table 3.2). When the DBMS is initiated after a failure, it starts by reading the `pg_control` file and then executes the WAL from the checkpoint record on. Therefore, if a disaster occurs after a checkpoint has been finished locally but not in the cloud, there will not be any problem because, although the WAL in the cloud may contain the last checkpoint record (writes WAL segments are handled by Algorithm 2), the `pg_control` file present in the cloud will not be up to date. As a result, during recovery PostgreSQL will start executing the WAL from the checkpoint record pointed by the `pg_control` file in the cloud (which may not be the last checkpoint record present in the WAL).

Going back to the algorithm, all the checkpoint synchronizations are performed in background by the *CheckpointThread* (Lines 17-20).

Recall from Section 4.5 that there are two types of DB objects that can be uploaded during checkpoints: *dumps* and *checkpoints*. During normal execution, GINJA keeps the write operations performed by the DBMS during a checkpoint and, as soon as the checkpoint is finished locally, submits those writes to be uploaded in one *checkpoint* DB object (Lines 6, 11-14). In this situation, all the write operations are aggregated in order to reduce the volume of data uploaded to the cloud (Line 6). On the other hand, whenever the total size of the DB objects in the cloud is greater or equal to 150% of the local database size, GINJA creates a new database *dump* and submits it to the *CheckpointThread* (Lines 8-10, 13-14). In this situation, GINJA will not execute any write in the DB objects while the *dump* object is being created, to guarantee that the database is dumped in a consistent way. This is not a problem because checkpoints are sporadic events that are performed in background (i.e., do not interfere with the execution of operations in the databases).

After uploading a new checkpoint, GINJA deletes objects that are no longer relevant from the cloud. This *Garbage Collection* is performed to reduce monetary costs (recall that cloud providers charge for the volume of storage used). We are aggressive in removing cloud objects as the *DELETE* operation is free in all clouds we are aware of.

Every time a DB object is completely uploaded to the cloud, GINJA removes all WAL objects with timestamps smaller than the current DB object uploaded since the last checkpoint (Lines 3-4 and 21-23). This is safe as such WAL objects contain information that will not be used in a recovery situation (i.e., WAL records previous to a WAL checkpoint record). Additionally, when the DB object uploaded is a *dump*, all the previous DB objects (including both incremental checkpoints and other dumps) are deleted as well (Lines 24-27).

Algorithm 3 Checkpoints and Garbage Collection.

```
Variables:
 1: checkpointQueue ← ∅
 2: timestamp ← ∅
Upon write(dbFile, offset, content) do:
 3: if ⟨dbFile, offset, content⟩ is the first write in checkpoint then
 4:     timestamp ← cloudView.getLastWALts()
 5: writeLocally(dbFile, offset, content)
 6: dbObject ← addAndAggregate(⟨dbFile, offset, content⟩)
 7: if ⟨dbFile, offset, content⟩ is the last write in checkpoint then
 8:     if cloudView.getTotalDBSize() ≥ 150% × local DB size then
 9:         dbObject ← create dump from local DB files
10:         dbObject.type ← "dump"
11:     else
12:         dbObject.type ← "checkpoint"
13:     dbObject.ts ← timestamp
14:     checkpointQueue.add(dbObject)
15:     dbObject ← ∅
CheckpointThread:
16: loop
17:     wait until checkpointQueue.size > 0
18:     obj ← checkpointQueue.remove()
19:     cloud.PUT("DB/"+obj.ts+"_"+obj.type+"_"+obj.size, obj)
20:     cloudView.addDB(obj.ts, obj.type, obj.size)
21:     for each walObject in cloudView such that walObject.ts < obj.ts do
22:         cloud.DELETE(walObject.objectName)
23:         cloudView.delete(walObject)
24:     if obj.type = "dump" then
25:         for each dbObject in cloudView such that dbObject.ts < obj.ts do
26:             cloud.DELETE(dbObject.objectName)
27:             cloudView.delete(dbObject)
```

4.7 Final Considerations

In this chapter we presented in detail the design of our solution, which seeks to reduce the monetary and management costs, while adding as few performance overhead as possible. To meet such objectives we have studied how DBMS manage data and how storage cloud providers charge for their services. This information was used to design and optimize our data and configuration model, as well as our algorithms.

The key design decisions covered here include: implementing GINJA at the file system level (which brings great benefits in terms of portability), using cloud storage services (as they allow the conception of very low-cost backup systems if used properly), creating a data model that makes a cost-efficient use of such services, and finally devising a powerful configuration system allows our users to have a tight control over the durability, performance and monetary costs of GINJA.

In the next chapter we will present some details about the implementation of our prototype.

Chapter 5

Implementation

In this chapter we cover the most relevant details behind the implementation of GINJA. We will start by presenting a few general considerations on the implementation and configuration of GINJA. Afterwards, we explore how GINJA is integrated with the DBMS, and present its architecture. Finally we include an UML diagram that addresses the structure of GINJA's code, and close this chapter with a few additional considerations.

5.1 General Considerations

All the software components that make up GINJA were implemented in JAVA within approximately 3700 lines of code distributed in 30 files. Table 5.1 states the number of lines of code that make up each module of GINJA.

It is possible to observe that GINJA is implemented in a generic way, as all the DBMS specific processing is performed in the PostgreSQL processor, which is a very small module (only 200 lines of code).

The largest module is the *Initialization*, which parses the system configuration, executes Algorithm 1, and initializes all the data structures and threads. The second largest module is the *Commit mechanism*, which implements the behaviour specified by the con-

Module	Lines of Code
Initialization	808
File System Interpreter	544
PostgreSQL Processor	200
Commit Mechanism	753
Checkpoint Mechanism	416
Cloud Synchronization	248
Cloud View	254
Other Data Structures	516
Total	3739

Table 5.1: Number of lines of code in each module of GINJA.

figuration parameters *B* and *S*. Note that the *File System Interpreter* also has a considerable amount of code, as it implements all the operations performed by the file system on the local storage device. Lastly, it should be mentioned that the *Cloud Synchronization* module is minimal because we use an external library to execute cloud operations (specifically we use DepSky's cloud storage drivers [31]).

The configuration parameters of GINJA are described in Table 5.2. The first two parameters are relative to the integration of GINJA (which will be covered in the next section), the following four are relative to our configuration model (described in Section 4.4), and the remaining ones cover aspects relative to the cloud synchronizations.

To perform compression we used the Deflater JAVA class, configured with the fastest compression level (for performance reasons). Regarding encryption, we used the JAVA Cipher class to encrypt data using the AES algorithm.

Parameter	Description
`pg_old`	Directory where the DMBS used to store its data before being integrated with GINJA.
`pg_new`	Directory where GINJA is mounted.
`B`	B system parameter (recall Table 4.1).
`tB`	T_B system parameter (recall Table 4.1).
`S`	S system parameter (recall Table 4.1).
`tS`	T_S system parameter (recall Table 4.1).
`nThreads`	Number of threads used to upload WAL objects to the cloud.
`mode`	Initialization mode. There are the following available modes: `init`, `reboot` and `recovery` (see Algorithm 1).
`cloud`	The cloud provider used to store data.
`cloudAccessKey`	Access key of the cloud service account.
`cloudSecretKey`	Secret key of the cloud service account.
`compression`	Whether the data should be compressed before uploaded.
`encryption`	Whether the data should be encrypted before uploaded.
`encryptionKey`	Key used to perform encryption.

Table 5.2: Configuration parameters of GINJA.

5.2 Integration of GINJA with the DBMS

GINJA is implemented as a specialized file system that intercepts the calls issued by the database management system and backs up the relevant information to a cloud storage service.

In order to operate correctly, GINJA needs to access the data managed by the DBMS (which in PostgreSQL is located in the `PGDATA` directory). However, we chose not to mount GINJA directly in this location. Instead, our mount point will be an empty directory in which the database directory will be mapped, i.e., all the system calls that are to be

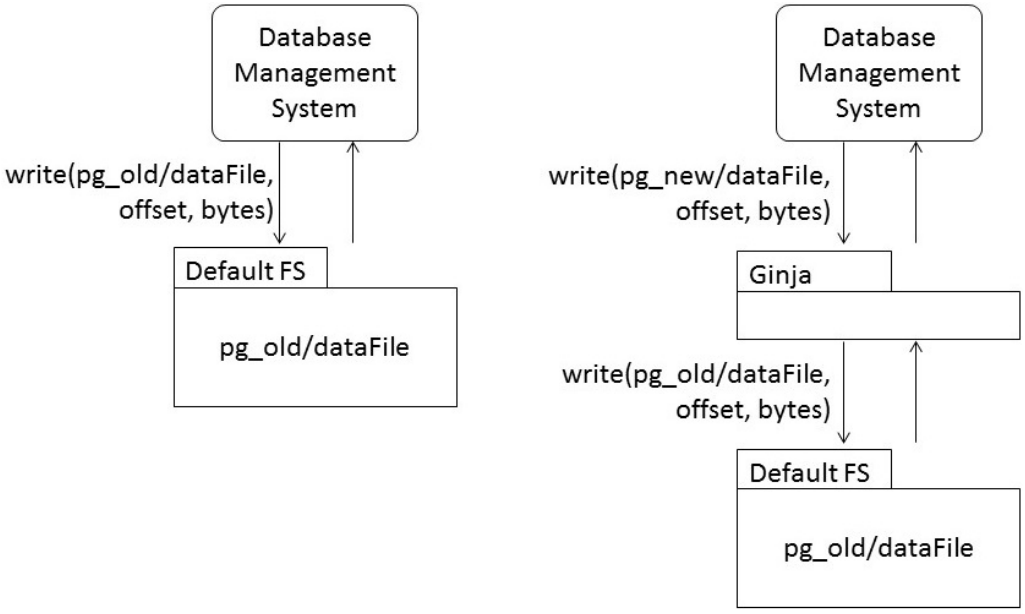

Figure 5.1: Interaction between PostgreSQL and the file system. The image on the left shows the DBMS using the native file system. In the right it is possible to observe the DBMS using GINJA.

issued in our mount point are performed in the `PGDATA` instead. This way, our system will be able to access the data that was previously in the DBMS.

Afterwards, the DBMS has to be configured to change its data directory to the one in which GINJA was mounted, so that its subsequent operations can be intercepted by our file system. This step requires a reboot of the DBMS, which reduces slightly the availability of the system.

Figure 5.1 shows how a DBMS interacts with the file system normally (in the left), and how a DBMS interacts with the file system when integrated with GINJA (in the right).

5.3 Architecture

Figure 5.2 presents the internal architecture of GINJA. Let us now briefly explain each of its components.

FS Interpreter. We implemented GINJA as a file system in user space using the FUSE (File system in USEr space) framework [7, 43]. FUSE is a kernel module that registers itself with the Virtual File System (VFS) as a regular file system, and communicates with a user space library. This library can be used by user space applications to implement file systems without performing modifications to the kernel.

As FUSE is implemented in C, we used a JAVA implementation of FUSE called FUSE-J [44]. This system is basically an API that uses Java Native Interface (JNI) bindings to FUSE and enables writing Linux file systems in JAVA language.

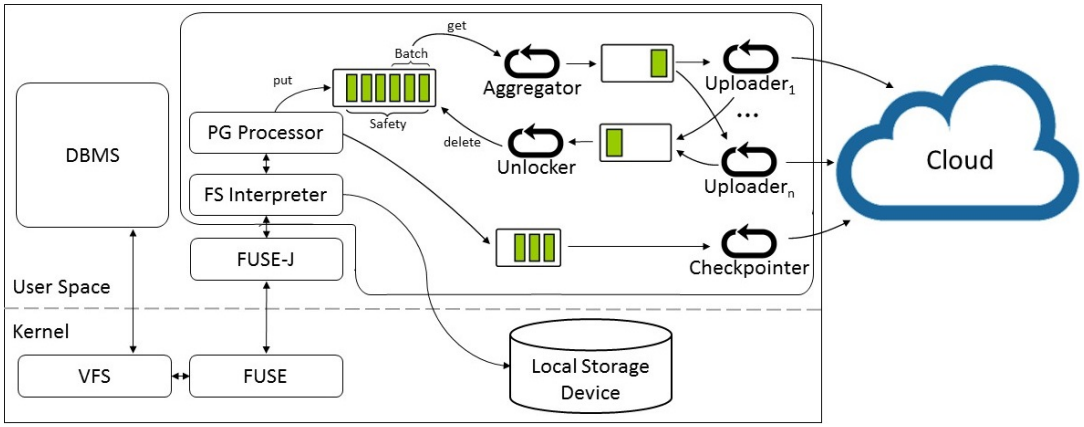

Figure 5.2: Detailed architecture of GINJA.

The FS Interpreter module presented in Figure 5.2 is an implementation of the FUSE-J interface *Filesystem3*. This module simply acts as a regular file system except that it performs the indirection explained in Section 5.2, and passes the file system calls received to the processor.

Processors. In order to make our DBMS Disaster Recovery Solution as generic and portable as possible, we have created the concept of *Processor*.

The *Processor* is a JAVA abstract class that contains all the operations defined in *Filsesystem3*. Each processor extension corresponds to a specific database management system, and is responsible for converting the file system calls received to a generic data format used by the remaining components of GINJA. 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 internals of the DBMS (specifically of how the data is managed at the file system level), which can take more effort.

In the scope of this project we have implemented only one processor, which is relative to PostgreSQL. However, a MySQL processor is currently being developed at LaSIGE.

Internals. As it can be observed in Figure 5.2, the processor uses two different queues to put the data received from the file system: one to treat the WAL writes and other to treat the checkpoint writes.

The write operations performed in the Write-Ahead Log (WAL) are sent to a specialized 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 the operation blocking.

A thread called *Aggregator* is responsible for getting sets of *B* write operations from this queue (without removing them), aggregating those writes into a single object for each WAL segment, and submitting the resulting data in a second queue.[1] A number of *Uploader* threads will retrieve elements from this queue and upload them in parallel as WAL objects, submitting acknowledgement to a third queue whenever a cloud upload completes. At last, a thread called *Unlocker* will remove sets of *B* elements from the head of *CommitQueue*, according to the acknowledgements received by the *Uploader* threads.

The write operations performed during checkpoints are submitted to a thread called *Checkpointer* that aggregates the data received and uploads it to the cloud in the form of DB cloud objects.

Note that Algorithm 2 is implemented in the *Aggregator*, *Uploader* and *Unlocker* threads, whereas Algorithm 3 is implemented in the *Checkpointer* thread. The Algorithm 1 is implemented in an initialization module not included in Figure 5.2.

5.4 Class Diagram

Figure 5.3 presents the UML class diagram of GINJA. The diagram was cleaned to present only the most important classes of the system.

FSInterpreter is an implementation of the FUSE-J's interface *Filesystem3*. This class performs the file system calls locally, and passes them to the *Processor*. The *Processor* performs a DBMS aware processing to those calls, submitting the results in the form of *FileWrite* instances to the *CommitQueue* and *CheckpointQueue*. GINJA's threads then take care of aggregating and uploading the data present on these queues as described in the previous section.

CommitQueue is the class that implements the coordination between the file system and the threads, by ensuring that the *B* and *S* parameters are respected (specifically, this queue uses the *TaskController* class to make sure that the effects of T_B and T_S are fulfilled).

The *Synchronizer* class is responsible for performing all the cloud synchronizations. It uses the implementation of the interface *IDepSkyDriver* (obtained from DepSky [31]) that corresponds to the desired cloud provider, and uses the classes *Compressor* and *Cipher* to compress and encrypt the data, respectively.

Finally, note that the *cloudView* data structure used to keep track of the objects existent in the cloud (recall Section 4.6) is divided in two classes: *WalView* and *DbView*. *WalView* is relative to the WAL objects, whereas *DbView* tracks the DB cloud objects. These classes are kept up to date by the *AggregatorThread* and the *CheckpointerThread*.

[1] This aggregation process reduces substantially the amount of information uploaded to the cloud by discarding all the data that was overwritten.

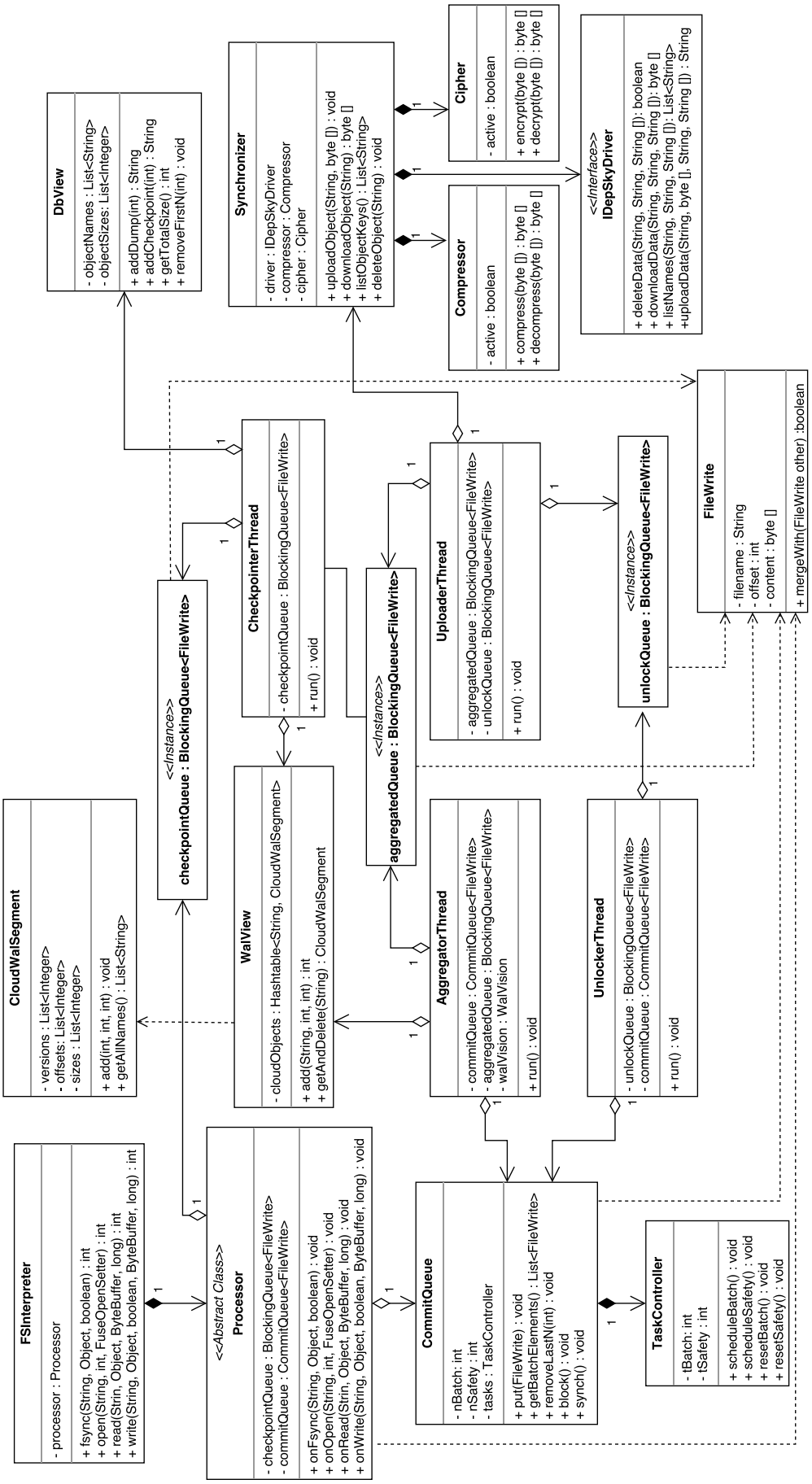

Figure 5.3: UML class diagram.

5.5 Final Considerations

In this chapter we have presented some details about the implementation of GINJA. We started by explaining how the integration and architecture of GINJA works, then we presented two diagrams that cover the structure of our code.

Although most of the details covered here may seem straightforward, they result from a long iterative process of analysis, design and evaluation. This process allowed us to create a modular prototype capable of performing disaster recovery for PostgreSQL in a cost-efficient way, as will be demonstrated in the next chapter.

Chapter 6

Evaluation

This chapter presents a detailed evaluation of the disaster recovery system developed in this project. In Section 6.1 we start with an analytical evaluation of GINJA in terms of monetary costs. Then, in Section 6.2, we present an experimental evaluation of our prototype that covers topics such as performance and cloud usage. Finally, in Section 6.3 we report the conclusions that can be drawn from the obtained results.

6.1 Economical Evaluation

The only element that introduces monetary costs to our solution is the use of cloud storage services. The prices of such services using different cloud providers are presented in Table 6.1. It is evident that the pricing model of this kind of services considers the volume of data in the cloud, the number and type of operations executed, and the bandwidth of data transfers. As a result, an economic evaluation of GINJA must consider these three factors.

Regarding the database size, we are going to explore how GINJA's data model influences the amount of storage used in the cloud. Performing an economical evaluation in terms of the cloud operations executed is way more complex. It involves studying the relation between the algorithms described in Section 4.6 and the performed operations. In terms of bandwidth, cloud providers only charge for the download traffic, which is used by GINJA only during recovery. As disasters are rare events, we focus this evaluation in GINJA's normal operation. For this reason, the bandwidth is not considered in this evaluation.

Storage Service	**Storage**	**Bandwidth**		**Operations**			
		Upload	Download	PUT	LIST	GET	DEL
Amazon S3	30 000	0	90 000	10	10	1	0
Azure Blob Storage	24 000	0	87 000	5	5	0.4	0
Google Cloud Storage	26 000	0	120 000	10	10	1	0

Table 6.1: Pricing of cloud storage services in microdollars (10^{-6}) . The storage and bandwidth prices are charged monthly per-GB, whereas the operational costs are relative to each executed operation.

It should be noted that for this evaluation it is necessary to identify the factors that influence the usage of the cloud. Such factors can be either related to GINJA (e.g., the configuration used) or to the database system itself (e.g., workload, size of the database).

6.1.1 GINJA Cost Model

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

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

Let us now explore in detail how each of the four components that make up this equation can be calculated.

Storage of DB Objects. As covered in Section 4.5, GINJA uploads the information of the database files in the form of DB objects. The storage cost relative to this kind of objects is given by the following formula:

$$Cost_{DB\_Storage} = DB\_Size \times 125\% \times Comp \times StorageCost$$

The DB_Size is measured in GB and the $StorageCost$ in \$/GB per 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, in average, the amount of DB data in the cloud will be 25% greater that the database size. Additionally, the amount of DB data uploaded can be reduced by using compression (represented as $Comp$), which decreases the value of $Cost_{DB\_Storage}$.

PUT Operations of DB Objects. The second factor that influences the total cost of our DR solution is the number of *PUT* operations used to upload DB objects. This depends essentially on how often the checkpoints occur, the average checkpoint size, and the price of each PUT operation. The cost of this component can be calculated as follows:

$$Cost_{DB\_PUT} = \frac{30 \times 24 \times 60}{Ckpt_{Period}} \times \left\lceil \frac{Ckpt\_Size}{1GB} \right\rceil \times PUT\_Cost$$

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. Recall that, in our data model, the maximum DB object size is 1GB. Consequently, for checkpoints greater than this amount of data, GINJA uploads several DB objects to the cloud.

Storage of WAL Objects. The third cost factor considered is relative to the volume of the WAL objects present in the cloud, and can be calculated as follows:

$$Cost_{WAL\_Storage} = \left(\left\lceil \frac{Workload \times Ckpt_{Time}}{RecordsPerPage} \right\rceil + 1 \right) \times PageSize \times Comp \times StorageCost$$

The first part of the equation determines the maximum number of WAL pages that can be in the cloud at any moment. Recall that all the WAL objects previous to a checkpoint are deleted from the cloud as soon as that checkpoint is completely uploaded. Consequently, the amount of storage is directly proportional to the number of updates per minute ($Workload$ – assuming that each update uses a record), and to the $CkptTime$, which includes the checkpoint period, plus its duration, plus the time it takes to be uploaded to the cloud.

The total number of updates performed between checkpoints is divided by the number of records per page ($RecordsPerPage$), reaching the number of WAL pages uploaded to the cloud. The "+1" covers the worst case scenario – the situation in which the first WAL write after a checkpoint is performed in the ending of a WAL page.

Finally, $PageSize$ is the size in GB of each WAL page, and $Comp$ represents the compression rate (i.e., the percentage of data reduced through compression).

PUT Operations of WAL Objects. Finally, the cost associated with the number of *PUT* operations of WAL cloud objects is represented by $Cost_{WAL\_PUT}$. This cost depends essentially on the database workload and the value of the *B* parameter, and it is given by the following formula:

$$Cost_{WAL\_PUT} = \frac{Workload \times 60 \times 24 \times 30}{B} \times PUT\_Cost$$

Every time *B* database updates are executed in the DBMS, a WAL object is uploaded to the cloud. Thus, $Cost_{WAL\_PUT}$ is obtained by calculating the number of database updates executed per month (note that $Workload$ is measured in requests per minute) and multiplying this value by the price charged for each *PUT* operation.

Discussion. We have analysed in detail all the components that influence the operational cost of GINJA.

The element that has major influence in the monetary cost of our solution is the execution of *PUT* operations of WAL objects. The cost of storing DB objects in the cloud can also be considerable, but only when considering very large databases. As GINJA targets small to medium sized databases, we expect the value of $Cost_{DB\_Storage}$ to be very low.

The remaining components considered result in negligible monetary costs. The cost of uploading DB objects will be minimal, as our system executes very few *PUT* operations per checkpoint (generally only one), and checkpoints are sporadic events in database

management systems. The cost of storing WAL objects in the cloud is also minor, as some of these objects are deleted from the cloud after a checkpoint is issued.

When recovering from a disaster, the monetary cost of GINJA depends on the number of objects downloaded and the volume of data present in those objects.

6.1.2 Evaluation

Figure 6.1 presents the operational monetary costs of GINJA (without compression) 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 that contain 75 WAL records. We also consider that a checkpoint happens every 60 minutes, and have a duration of 20 minutes.

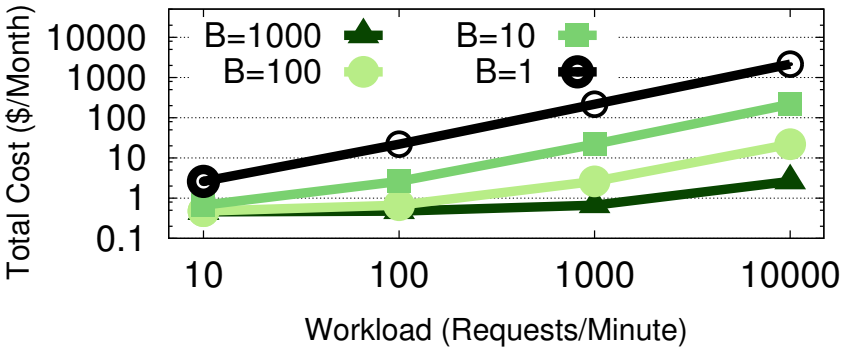

Figure 6.1: Effect of different configurations and workloads in GINJA's monetary cost considering Amazon S3.

It is possible to note that, as expected, the B parameter has a severe impact on the total monetary cost of our solution. This can be explained by the fact that B reduces the number of *PUT* operations executed in the cloud. Additionally, we can observe that this relation is even more evident when considering large workloads. Nevertheless, note that there are plenty of possible configurations that result in a total cost of less than to $3 per month.

Let us now present an evaluation of a real use case, considering two databases used in a clinical analysis system.

Table 6.2 presents the monetary costs of performing disaster recovery in the cloud (specifically Amazon Web Services) using GINJA, and using database replication with virtual machines.[1] We consider two database configurations: one hospital with a 1TB of data and a workload of 840 updates per minute, and a Medical Laboratory with a 10GB-database that processes 48 updates per minute.

In the hospital scenario, GINJA has a cost approximately $4\times$ smaller than the cost of using virtual machines. Note that in this situation the major portion of our cost is relative

[1] Values calculated using `https://calculator.s3.amazonaws.com/index.html`.

Configuration	GINJA	VMs in Amazon
Hospital (1TB, 840 up/min)	**$39.7** (data loss < 1 min) **$63.4** (data loss < 5 sec)	38 (VM t2.medium) +116 (EBS 500IOS) +36.6 (VPN) = **$190.6**
Laboratory (10GB, 48 up/min)	**$2.5** (data loss < 1 min) **$10.7** (data loss < 5 sec)	19 (VM t2.small) +18 (EBS 100IOS) +36.6 (VPN) = **$61.6**

Table 6.2: Costs of performing cloud-based disaster recovery with AWS using GINJA or database replication with VMs.

to the DB objects maintained in the cloud infrastructure. In the second scenario, GINJA presents a cost gain of $25\times$ (with a maximum data loss of one minute) and $6\times$ (with a maximum data loss of 5 seconds) in relation to running VM instances in the cloud. Here, the dominant cost is originated by uploading WAL objects to the cloud. It should be noted that besides the economical advantages presented, our system is also way easier to manage than the alternative solution, which requires configuring a firewall, setting up a public IP and so forth.

6.2 Experimental Evaluation

In this section we present a detailed experimental evaluation of GINJA using the PotgreSQL 9.3 database management system [18]. The experiments were executed in two Dell Power Edge R410 machines equipped with two Intel Xeon E5520 CPUs (quad-core, HT, 2.27Ghz), 32GB of RAM and a 146GB Hard Disk Drive with 15k RPMs. The operating system used was Ubuntu Server Precise Pangolin (12.04 LTS, 64-bits), with kernel 3.5.0-23-generic and Java 1.8.0 (64-bits). The cloud storage service used was Amazon S3 (US Standard).

The results presented come from the average of five executions, applying a TPC-C workload [45] during 5 minutes, using the BenchmarkSQL 4.1.1 tool [46]. The metrics observed were the total number of transactions per minute (Tpm-Total), and the number of a specific type of update transaction (newOrder) processed per minute while the DBMS is processing other types of transactions (Tpm-C).

6.2.1 Performance

As we have previously described, GINJA performs cloud synchronizations in parallel. One factor that influences the performance of our system is the number of threads used to perform such synchronizations.

Figure 6.2 shows the total number of transactions per minute achieved by GINJA for different values of B and S, and number of uploader threads. The results show that, for all the configurations used, increasing the number of threads raises the throughput of

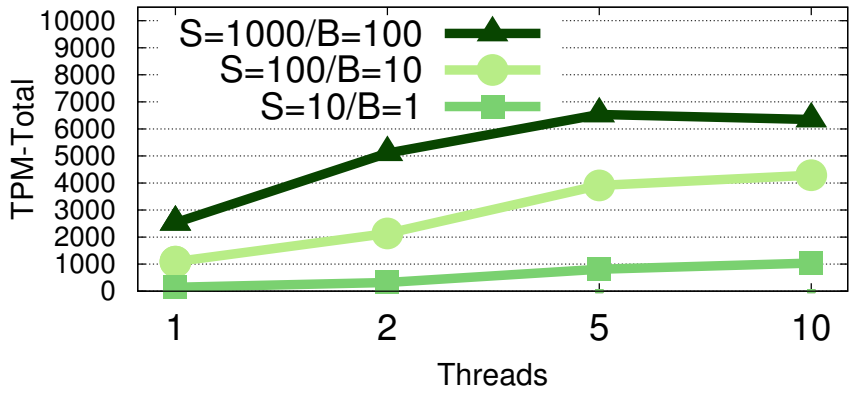

Figure 6.2: Influence of the number of threads in GINJA's throughput.

the database management system. However, this benefit is not significant when using more than 5 threads. These results are consistent with existing literature [47]. For this reason, we used 5 threads to perform cloud synchronizations in all the other experiments presented in this evaluation.

Figure 6.3 shows the effect that different configurations of B and S have in the throughput of the DBMS. Using the native file system, the DBMS processes a total of 8290 transactions per minute. When using a similar file system implemented with FUSE, the database system decreases its throughput by 7%, achieving a total of 7690 transactions per minute. As GINJA is implemented using FUSE, this will be our baseline.

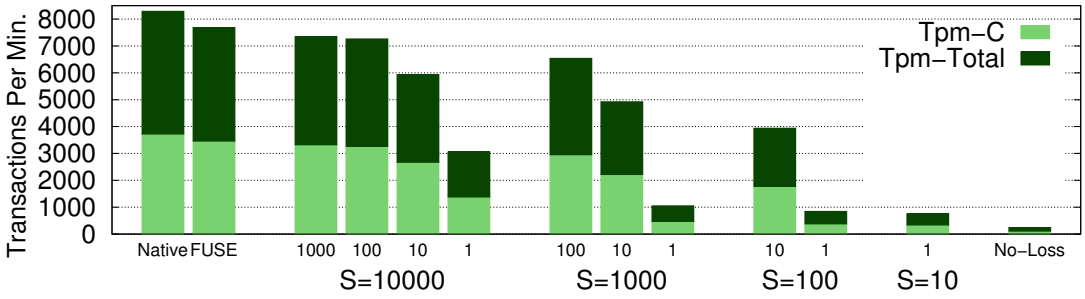

Figure 6.3: Influence of different configurations in the performance of GINJA. The values of B are expressed immediately below the columns. Exceptions are the first two columns, which are relative to the native file system and to FUSE, and the last column, which reefers to the configuration: $B = 1$, $S = 1$.

The figure clearly shows that the B and S parameters have an influence in the total amount of transactions per second that the system can process. The value of S has an impact on GINJA's performance because, when this parameter is reached, the DBMS blocks. B also has an effect on this matter because GINJA uses a finite number of threads to upload WAL objects, each one containing B database updates. For this reason, using small values of B causes the S parameter to be reached earlier, which decreases the performance

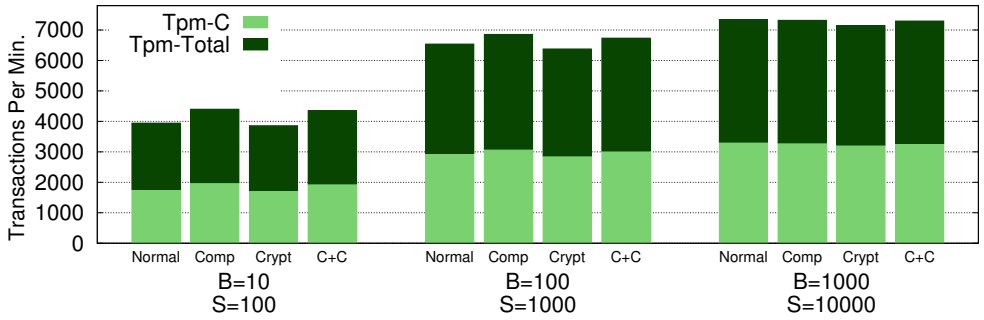

Figure 6.4: Effect of compression and cryptography in the performance of GINJA. The columns are grouped by configuration, and the values immediately below de columns specify whether Compression and Cryptography are active (in the columns "C+C" both compression and cryptography are active).

of the DBMS. The results also show that, for high values of *B* and *S*, GINJA introduces a minimal performance overhead (in the order of 4.5%).

Note that it is possible to use GINJA while guaranteeing a durability level of 100%. This corresponds to *synchronous replication*, and can be achieved by using the configuration: B=1, S=1. As expected, this configuration brings a substantial performance degradation that can be observed in the last column of Figure 6.3, with a Tpm-Total value of 242.

Figure 6.4 shows how compression and encryption influence the performance of our system. First, it is evident that compression increases the throughput of GINJA. This is because it reduces the amount of data uploaded to the cloud, which causes a decrease in the synchronization latency (this will be covered in Section 6.2.2). Second, note that encryption introduces a minimal overhead in the performance of GINJA. This can be observed by comparing the columns "Normal" and "Crypt", and the columns "Comp" and "C+C".

It should be noted that, even though we have presented the values of Tpm-Total and Tpm-C in Figures 6.3 and 6.4, our discussion only mentioned the metric Tpm-Total. This is because, in our results, these two metrics present a similar behaviour.

6.2.2 Cloud Usage

Table 6.3 shows the number of *PUT* operations executed, the size of the objects written and the latency of this cloud operation.

The results show that increasing the *B* parameter from 10 to 100 decreases the number of *PUT* operations performed during a TPC-C execution by 80%, while an additional tenfold increase in *B* further decreases this number by almost 70%. In the same way, increasing *B* increases the object size and, consequently, the latency to write objects to the cloud infrastructure. However, due to the page coalescing of the system, the observed increase in the size of the objects is smaller than the *B* increase.

Configuration (B/S mode)	**Num. PUTs** (TPC-C, 5 min)	**Object Size** (kB)	**PUT latency** (milliseconds)
10/100 plain	1789	386	692
10/100 Comp	2002	239	549
10/100 Crypt	1767	387	592
10/100 C+C	1990	237	562
100/1000 plain	364	3018	2880
100/1000 Comp	383	1903	1932
100/1000 Crypt	354	3002	3198
100/1000 C+C	383	1908	2007
1000/10000 plain	119	10081	7707
1000/10000 Comp	118	6366	6328
1000/10000 Crypt	111	10043	9494
1000/10000 C+C	119	6339	4422

Table 6.3: GINJA's storage cloud usage. All results are average collected during five executions of five minutes of TPC-C, with standard deviations under 12%.

The table also shows the consequences that the encryption and compression modes have in GINJA's cloud usage. It is possible to observe that using compression reduces the volume of data uploaded in each cloud object, which reduces the latency as a consequence. This results in the performance benefits discussed before. It is also evident that using compression and encryption leads to more *PUT* operations executed, which is due to the fact that such configuration achieve higher throughput levels.

6.2.3 Database Server Resource Usage

Table 6.4 presents the resource usage of a PostgreSQL server serving a TPC-C workload under different configurations with and without GINJA.

The table shows that using a Native or FUSE-J file system already requires around 7% of the machine CPU and less than 1.6GBs of memory ($< 5\%$). When adding GINJA's disaster recovery FS, the server CPU and memory usage increase 1% and 2%, respectively,

Configuration	**CPU**	**Memory**
Native FS	6.4%	4.3%
FUSE-J FS	6.9%	4.9%
100/1000	7.8%	6.9%
100/1000 Comp	11.6%	9.7%
100/1000 Crypt	9.1%	7.2%
100/1000 C+C	13.4%	9.9%

Table 6.4: PostgreSQL server (eight cores with hyper-threading and 32GB of RAM) resource usage with and without GINJA.

when compared with a FUSE-J FS, mostly due to the queues, buffers and data structures used in its implementation. Additionally, compression and cryptography introduce significant CPU usage: +3.8% and +1.3%, respectively. In terms of memory, these features increase the memory usage by 2.8% (compression) and 0.3% (encryption). When compression and encryption are used together, the overheads of these features are summed up.

In the end, using GINJA with compression and encryption requires +7% of CPU (less than a core in our server) and +5.6% of memory (less than 2GB) of our 8-core server and 32GB of memory. We consider these costs would not be a deterrent for using GINJA.

6.2.4 Recovery Time

Now we are going to present our last experiment, which evaluates the recovery time of our DR solution. To do these experiments we executed a TPC-C workload for five minutes, and then injected a fault. After that, we executed GINJA in recovery mode and measured the time it took to download and reconstruct the database. We conducted this experiment for three different database sizes (by varying the number of warehouses in TPC-C [45]) and executed the recovery process in a machine located in our lab (in Lisbon), and in an Amazon EC2 virtual machine (located in the same region where GINJA's data is backed up).

Figure 6.5 shows that the recovery time grows with the database size. This is expected, as recovering bigger databases involves downloading a greater volume of data. It is also evident that the recovery time can be significantly reduced by executing GINJA in a computing instance located in the same cloud infrastructure where the data is. Although we consider these results adequate, they can be improved by reducing the maximum DB object size and using several threads to download cloud objects in parallel.

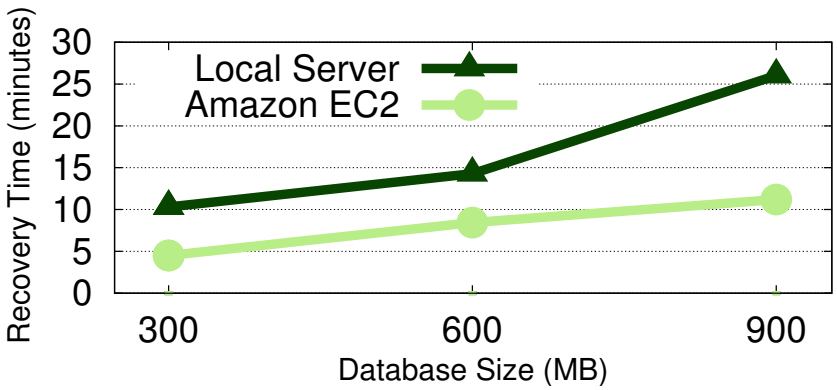

Figure 6.5: Recovery times of GINJA for different database sizes using a local server and a VM instance in the cloud.

6.3 Discussion

In this chapter we presented an in depth evaluation of GINJA covering topics such as monetary costs, performance, and resource utilization.

The economical evaluation shows that GINJA allows database management systems to perform cloud-based disaster recovery with very low associated monetary costs (e.g., a 10GB database with a workload $\leq$ 48 updates/minute can tolerate disasters with less than one minute of data loss for only \$0.9). The main factor that introduces costs during normal operation is the upload of WAL objects to the cloud. This cost can be reduced by choosing configurations with higher B values. The storage of cloud DB objects can also introduce significant costs, but only for large databases.

Regarding the experimental evaluation, we concluded that GINJA is capable of performing disaster recovery with very low performance overhead if properly configured. We have analysed the consequences that the parameters B and S have in GINJA's cloud usage. Additionally, we have showed that compression can bring some performance benefits by reducing the volume of data uploaded to the cloud (besides lowering the synchronization latency, this also reduces the monetary costs relative to storing objects in the cloud) at the expense of increasing the resource consumption in the database server. We have also demonstrated that using cryptography to ensure confidentiality introduces a minor overhead in the performance of the database management system. Lastly we showed that GINJA's recovery time is quite large, but can be reduced by using a computational instance (such as virtual machine) located in the same cloud infrastructure used by GINJA to backup its data.

Chapter 7

Conclusion

In this thesis we presented GINJA: a low-cost disaster recovery system for database management systems that relies entirely on cloud storage services to backup its data.

The main factors that differentiate GINJA from the existing DR solutions are: very low monetary cost, fine-grained control over the data that can be lost during disasters, high level of portability, and low performance overhead under failure-free operation. We coped with such challenges by:

1. Studying the internal functioning of the database management systems, with special focus on it interaction with the file system;

2. Understanding the pricing model of cloud storage services;

3. Creating a configuration model that allows our users to define precisely the levels of cost, durability and performance, in accordance with their application requirements;

4. Devising a data model that allows an efficient use of the cloud storage services both in terms of performance and monetary costs;

5. Developing the necessary algorithms to perform disaster recovery in a safe way, based on all the information covered in the previous items;

6. Implementing these algorithms at the file system level (for portability reasons).

We have conducted a series of experiments that seek to evaluate GINJA in terms of performance, monetary costs, resource usage and recovery time. The results obtained from our evaluation show that it is possible to implement disaster recovery for DBMS in a safe, cheap and efficient way, relying exclusively on cloud storage services. Our evaluation also show the effects caused by GINJA's features (e.g., compression and encryption) and configuration parameters in the resulting throughput and monetary cost

7.1 Future Work

As future work we plan to extend GINJA to support other database management systems. Although this can be easily achieved by implementing new processors, it requires an in depth understanding of the internals of other DBMSs.

Additionally, we intend to create a module capable of performing an autonomic configuration of parameters such as the number of threads used to upload data, and how many database updates are included in each cloud synchronization (i.e., the *B* parameter). In this way, the system administrator would only have to define the desired level of durability (i.e., the *S* parameter), and GINJA would dynamically adjust the remaining parameters in the most efficient way possible.

Finally, the software developed in this project will be a key demonstration in the intermediate review of the SUPERCLOUD H2020 project. In this demonstration, our system will be integrated with CLINIDATA (MAXDATA intelligent management system for clinical laboratories). For this reason, we will test GINJA extensively with the database schemas used by this application, and prepare an integrated demonstration of our product, which will be presented next September.

Bibliography

[1] Kimberly Keeton, Cipriano A Santos, Dirk Beyer, Jeffrey S Chase, and John Wilkes. Designing for disasters. *FAST*, 2004.

[2] Glen Robinson, Ianni Vamvadelis, Attila Narin, et al. Using amazon web services for disaster recovery. 2011.

[3] Timothy Wood, Emmanuel Cecchet, KK Ramakrishnan, Prashant Shenoy, Jacobus Van Der Merwe, and Arun Venkataramani. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *2nd USENIX workshop on hot topics in cloud computing*, pages 1–7, 2010.

[4] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. Remusdb: Transparent high availability for database systems. *The VLDB Journal*, 22(1):29–45, 2013.

[5] Shriram Rajagopalan, Brendan Cully, Ryan O'Connor, and Andrew Warfield. Secondsite: disaster tolerance as a service. In *ACM SIGPLAN Notices*, volume 47, pages 97–108. ACM, 2012.

[6] Timothy Wood, H Andrés Lagar-Cavilla, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. Pipecloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 17. ACM, 2011.

[7] Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[8] Joel Alcântara, Tiago Oliveira, and Alysson Bessani. Ginja: Recuperação de desastres de baixo custo para sistemas de gestão de bases de dados. *INForum2016*, 2016.

[9] Business continuity statistics: Where myth meets fact, continuity central, 2009.

[10] On the quest for the mysterious source of the 'data loss causes company failure' statistic, IT Knowledge Exchange, 2014.

[11] Downtime and data loss cost enterprises $1.7 trillion per year: Emc, security week, 2014.

[12] Smb security and data protection: survey shows high concern, less action, symantec, 2009.

[13] Small Business IT Survey: No Backup, No Data, No Business, Small Business Computing, 2014.

[14] Business continuity trends and challenges, continuity central, 2016.

[15] Log-shipping standby servers on PostgreSQL. `https://www.postgresql.org/docs/current/static/warm-standby.html`.

[16] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. Snapmirror®: file system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 9–9. USENIX Association, 2002.

[17] Minwen Ji, Alistair C Veitch, John Wilkes, et al. Seneca: remote mirroring done write. In *USENIX Annual Technical Conference, General Track*, pages 253–268, 2003.

[18] PostgreSQL. `http://www.postgresql.org/`.

[19] MySQL. `https://www.mysql.com/`.

[20] Rafal Cegiela. Selecting technology for disaster recovery. In *Dependability of Computer Systems, 2006. DepCos-RELCOMEX'06. International Conference on*, pages 160–167. IEEE, 2006.

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

[22] Peter Brouwer. The art of data replication. *Oracle Technical White Paper*, 2011.

[23] Microsoft Azure Site Recovery. `https://azure.microsoft.com/en-us/services/site-recovery/`.

[24] VMware vCloud Air Disaster Recovery. `https://www.vmware.com/cloud-services/infrastructure/vcloud-air-disaster-recovery`.

[25] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine

replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[26] Dave Hitz, James Lau, and Michael A Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, 1994.

[27] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[28] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[29] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage (TOS)*, 5(4):14, 2009.

[30] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: a shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, 2014.

[31] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[32] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 251–264. ACM, 2008.

[33] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[34] Rudolf Bayer and Edward McCreight. *Organization and maintenance of large ordered indexes*. Springer, 2002.

[35] Michael Stonebraker and Lawrence A Rowe. *The design of Postgres*, volume 15. ACM, 1986.

[36] Jayadevan Maymala. *PostgreSQL for Data Architects*. Packt Publishing Ltd, 2015.

[37] Abraham Silberschatz, Henry F Korth, S Sudarshan, et al. *Database system concepts*, volume 6. McGraw-Hill Singapore, 2011.

[38] PostgreSQL Documentation. http://www.postgresql.org/docs/.

[39] strace UNIX man page.

[40] Victor A. Abell. lsof UNIX man page.

[41] MySQL - The InnoDB Storage Engine. http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

[42] Oracle Database. https://www.oracle.com/database/.

[43] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra incognita: On the practicality of user-space file systems.

[44] FUSE-J. http://fuse-j.sourceforge.net/.

[45] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[46] BenchmarkSQL. https://bitbucket.org/openscg/benchmarksql.

[47] Binbing Hou, Feng Chen, Zhonghong Ou, Ren Wang, and Michael Mesnier. Understanding i/o performance behaviors of cloud storage from a client's perspective.

[48] Fred B Schneider. What good are models and what models are good. *Distributed systems*, 2:17–26, 1993.

[49] William D. Norcott. iozone UNIX man page.

[50] Russell Coker. postmark UNIX man page.