Linux Backup Automation with rsync & Borg

Designing Secure, Efficient, and Fully Automated Backup Systems on Linux

Preface

Every Linux system administrator has a story about the backup that wasn't there when it mattered most. Perhaps it was a corrupted database on a production server, a misconfigured script that silently overwrote critical files, or a ransomware incident that turned months of work into encrypted rubble. In each case, the lesson is the same: **backups are only as good as the system that creates, verifies, and protects them**.

This book exists because backing up data on Linux shouldn't be an afterthought—it should be an engineered, automated, and battle-tested process. *Linux Backup Automation with rsync & Borg* is a practical guide to designing backup systems that run reliably without constant human intervention, leveraging two of the most powerful and trusted tools in the Linux ecosystem: **rsync** and **BorgBackup**.

Who This Book Is For

Whether you're a Linux system administrator managing a handful of servers, a DevOps engineer responsible for infrastructure reliability, or an enthusiast who wants to protect a personal Linux workstation, this book meets you where you are. The only prerequisites are a working familiarity with the Linux command line and a genuine desire to stop losing data.

What You'll Learn

The book is organized into a deliberate progression. We begin by establishing *why* backup automation matters and grounding you in the fundamental concepts—backup types, storage models, and the principles that separate fragile backup habits from resilient ones.

From there, we dive deep into **rsync**, exploring its core mechanics before building real-world incremental backup solutions with shell scripts, `cron` jobs, and `systemd` timers—the native scheduling tools of modern Linux distributions. You'll learn to push backups securely over SSH and encrypt data in transit, because a backup strategy without security is simply a liability waiting to be exploited.

The second major arc introduces **BorgBackup**, a deduplicating, encrypting backup tool purpose-built for the demands of Linux environments. You'll create and manage Borg repositories, automate backup workflows, and configure secure offsite storage—all while understanding the architectural decisions behind each choice.

The final chapters elevate the conversation from tactical execution to strategic thinking. We cover **monitoring and alerting**, because an unmonitored backup is an assumption, not a guarantee. We address **restore testing and validation**, because a backup you've never restored is a backup you can't trust. And we explore **enterprise backup policy design** and the evolving role of the Linux professional—from traditional system administrator to modern reliability engineer.

Five appendices provide quick-reference material you'll reach for repeatedly: an rsync option cheat sheet, a Borg command reference, backup rotation templates, sample automation scripts, and a disaster recovery checklist.

The Philosophy Behind This Book

Every recommendation in these pages has been shaped by a single conviction: **automation is not a luxury—it is the minimum standard for responsible data stewardship on Linux**. Manual backups fail because humans forget, get busy, and make mistakes. Automated backups, properly designed and monitored, do not. This book gives you the knowledge to build systems that embody that principle.

Acknowledgments

This book owes a debt of gratitude to the open-source communities behind rsync, BorgBackup, and the broader Linux ecosystem. Their tireless work has given all of us tools of extraordinary power and reliability—freely available, endlessly adaptable, and worthy of deep understanding. I'm also grateful to the countless system administrators, forum contributors, and technical writers whose shared experiences have shaped the best practices documented here.

Finally, thank you—the reader—for taking data protection seriously. The hours you invest in these chapters will pay dividends the first time a crisis strikes and your automated Linux backup system delivers exactly what it promised.

Let's build something resilient.

Bas van den Berg

Table of Contents

Chapter 1: Why Backup Automation Matters

Data is the lifeblood of every modern organization, every personal project, and every server that hums quietly in a data center somewhere. Yet despite this fundamental truth, an alarming number of Linux administrators, developers, and even seasoned engineers treat backup strategies as an afterthought. They configure their servers, deploy their applications, tune their databases, and then somewhere down the road, when disaster strikes, they realize that the one thing they neglected was the very thing that could have saved them. This chapter lays the foundation for everything that follows in this book by exploring why backup automation on Linux is not merely a convenience but an absolute necessity. We will examine the real cost of data loss, understand the philosophy behind automated backups, explore the Linux tools that make automation elegant and reliable, and establish the mindset that will guide you through the rest of your journey with rsync and Borg.

The Reality of Data Loss in Linux Environments

Before diving into any technical discussion, it is worth pausing to consider the scale of the problem. Data loss is not a hypothetical scenario reserved for textbooks and certification exams. It happens every single day, across every industry, on servers running every operating system, including Linux. In fact, because Linux powers the vast majority of the world's servers, cloud infrastructure, containers, and embed-

ded systems, the impact of data loss on Linux systems is disproportionately significant.

Consider the following scenarios, all of which are commonplace in Linux administration:

A system administrator accidentally runs `rm -rf /var/lib/mysql/` instead of targeting a specific subdirectory. The entire MySQL data directory vanishes in seconds. There is no recycle bin in the Linux command line. There is no "undo" button. The data is gone.

A ransomware attack encrypts every file on a web server running Ubuntu. The attacker demands payment in cryptocurrency. Without a recent, clean backup stored offsite, the organization faces an impossible choice between paying the ransom and losing everything.

A RAID controller fails silently on a CentOS server. The administrator assumed that RAID provided redundancy, which it does, but RAID is not a backup. When the filesystem becomes corrupted, the RAID array faithfully mirrors the corruption across all disks.

A developer pushes a flawed migration script to a production PostgreSQL database running on Debian. The script drops critical tables. The transaction was committed before anyone noticed. Without a point-in-time backup, the data from those tables is irrecoverable.

These are not edge cases. These are the everyday realities of working with Linux systems. The question is never whether data loss will occur but when it will occur and whether you will be prepared.

The following table summarizes common causes of data loss on Linux systems and their relative frequency and impact:

Cause of Data Loss	Frequency	Typical Impact	Preventable with Backups
Human error (accidental deletion, misconfigurations)	Very High	Moderate to Severe	Yes
Hardware failure (disk, controller, memory)	Moderate	Severe	Yes
Software bugs (application or kernel level)	Moderate	Moderate to Severe	Yes
Ransomware and malware attacks	Increasing	Critical	Yes, if backups are isolated
Natural disasters (fire, flood, power surge)	Low	Catastrophic	Yes, with offsite backups
Filesystem corruption	Low to Moderate	Severe	Yes
Theft of physical hardware	Low	Critical	Yes, with offsite or cloud backups

Every single entry in that table can be mitigated, and in many cases entirely recovered from, with a properly implemented and automated backup strategy. The key word there is "automated," and that distinction is what separates organizations that recover gracefully from those that do not recover at all.

The Difference Between Having Backups and Having Automated Backups

Many Linux administrators will tell you they have backups. And technically, they might. Perhaps once a month, someone manually runs a `tar` command to com-

press a directory and copies it to an external drive. Perhaps there is a script sitting in someone's home directory that was written two years ago and has not been updated since. Perhaps someone set up a cron job once, but the destination disk filled up six months ago and nobody noticed because there was no monitoring in place.

These are not backup strategies. These are backup intentions. The gap between intention and strategy is where data loss lives.

Manual backups fail for predictable, human reasons. People forget. People get busy. People leave the organization and take their institutional knowledge with them. People make mistakes when they do remember. The entire point of automation is to remove the human element from the repetitive, critical, and error-prone process of creating and managing backups.

Automated backups on Linux offer several fundamental advantages over manual approaches:

Consistency. An automated backup runs at the same time, in the same way, every single time. It does not have bad days. It does not get distracted. It does not decide to skip this week because the server "seems fine."

Reliability. When properly configured with error handling, logging, and monitoring, an automated backup system will alert you when something goes wrong. A manual process has no such mechanism. If a human forgets to run the backup, there is no notification that the backup was missed.

Efficiency. Tools like rsync and Borg are designed to perform incremental backups, meaning they only transfer or store data that has changed since the last backup. This makes frequent automated backups practical even for systems with large datasets. A human performing manual backups is far more likely to create full copies every time, wasting time, bandwidth, and storage.

Auditability. Automated systems produce logs. Logs provide an audit trail that tells you exactly what was backed up, when it was backed up, how long it took, and

whether any errors occurred. This audit trail is invaluable for compliance, troubleshooting, and capacity planning.

Recoverability. The ultimate purpose of any backup is recovery. Automated systems that are regularly tested ensure that when you need to restore data, the process is well-understood and the backups are known to be valid. Manual backups that are never tested provide a false sense of security that can be worse than having no backups at all.

Let us illustrate this with a simple comparison. Consider two approaches to backing up the `/etc` directory on a Linux server, which contains critical system configuration files.

The manual approach might look like this, executed by an administrator who remembers to do it:

```
tar -czf /backup/etc-backup-$(date +%Y%m%d).tar.gz /etc
```

This command creates a compressed tarball of the `/etc` directory with a date stamp in the filename. It works, but it depends entirely on a human remembering to run it. There is no error checking, no logging, no notification if it fails, and no mechanism to clean up old backups.

The automated approach, even at its simplest, looks fundamentally different:

```
#!/bin/bash
# /usr/local/bin/backup-etc.sh
# Automated backup of /etc directory

BACKUP_DIR="/backup/etc"
LOG_FILE="/var/log/backup-etc.log"
DATE=$(date +%Y%m%d-%H%M%S)
RETENTION_DAYS=30

echo "[$DATE] Starting /etc backup..." >> "$LOG_FILE"

mkdir -p "$BACKUP_DIR"
```

```
if tar -czf "$BACKUP_DIR/etc-backup-$DATE.tar.gz" /etc 2>>
"$LOG_FILE"; then
    echo "[$DATE] Backup completed successfully." >> "$LOG_FILE"
else
    echo "[$DATE] ERROR: Backup failed!" >> "$LOG_FILE"
    echo "Backup of /etc failed on $(hostname) at $DATE" | mail
-s "BACKUP FAILURE" admin@example.com
    exit 1
fi

# Clean up backups older than retention period
find "$BACKUP_DIR" -name "etc-backup-*.tar.gz" -mtime +
$RETENTION_DAYS -delete
echo "[$DATE] Old backups cleaned up. Retention: $RETENTION_DAYS
days." >> "$LOG_FILE"

exit 0
```

This script is then scheduled using cron, the time-based job scheduler that is a fundamental component of every Linux system:

```
# Edit the crontab
crontab -e

# Add the following line to run the backup every day at 2:00 AM
0 2 * * * /usr/local/bin/backup-etc.sh
```

The cron entry follows this format:

Field	Value	Meaning
Minute	0	At minute zero
Hour	2	At 2 AM
Day of Month	*	Every day of the month
Month	*	Every month
Day of Week	*	Every day of the week

Even this simple example demonstrates the qualitative difference between a manual backup and an automated one. The automated version logs its activity, handles errors, sends notifications on failure, manages retention automatically, and runs without any human intervention. And this is just a basic shell script. The tools we will explore in this book, rsync and Borg, provide far more sophisticated capabilities.

Note: The `crontab -e` command opens the current user's crontab file in the default text editor. For system-wide backup tasks, it is often more appropriate to place scripts in `/etc/cron.daily/`, `/etc/cron.weekly/`, or to use `/etc/crontab` directly. The choice depends on your specific requirements and organizational standards.

Why Linux Is the Ideal Platform for Backup Automation

Linux is not merely a platform on which backup automation can be performed. It is the platform for which backup automation was, in many ways, designed. The Unix philosophy that underlies Linux, the philosophy of small, composable tools that each do one thing well, is the very foundation of effective backup automation.

Consider the tools that are available natively or easily installable on virtually every Linux distribution:

rsync is a file synchronization tool that has been a cornerstone of Linux administration for decades. It uses a delta-transfer algorithm to send only the differences between source and destination files, making it extraordinarily efficient for incremental backups. It supports compression, SSH tunneling for secure remote transfers, bandwidth limiting, and fine-grained control over which files are included or excluded. It is installed by default on nearly every Linux distribution.

Borg (BorgBackup) is a deduplicating backup program that represents the modern evolution of backup technology on Linux. It provides encryption, compression, and deduplication at the chunk level, meaning that even if you back up the same file in multiple locations or across multiple backup runs, it is stored only once. Borg supports append-only repositories for ransomware resistance and provides excellent performance even with very large datasets.

cron and systemd timers provide the scheduling infrastructure. Cron has been the standard job scheduler on Unix and Linux systems for decades, and systemd timers offer a more modern alternative with better logging integration and dependency management. Both are available on every mainstream Linux distribution.

SSH provides the secure transport layer. With key-based authentication, Linux systems can communicate securely without passwords, enabling fully automated remote backups without any human interaction.

Shell scripting (Bash) ties everything together. The Bash shell, available on every Linux system, provides the glue that connects these tools into coherent, maintainable backup workflows. Variables, conditionals, loops, functions, error handling, and all the other constructs of a full programming language are available to the backup administrator.

The following table shows how these Linux tools map to the requirements of a comprehensive backup system:

Backup Requirement	Linux Tool	How It Addresses the Requirement
Efficient file transfer	rsync	Delta-transfer algorithm sends only changes
Data deduplication	Borg	Chunk-level deduplication across all backups

Encryption at rest	Borg	AES-256 encryption of repository data
Encryption in transit	SSH	Encrypted tunnel for all remote operations
Scheduling	cron / systemd timers	Time-based execution of backup scripts
Compression	rsync (flag), Borg (built-in), gzip, zstd	Multiple compression options at various levels
Logging and monitoring	syslog, journalctl, custom log files	Comprehensive logging infrastructure
Notification on failure	mail, sendmail, custom webhook scripts	Multiple notification mechanisms
Retention management	find (with cron), Borg prune	Automated cleanup of old backups
Remote backup	rsync over SSH, Borg over SSH	Native support for remote repositories

No other operating system provides this level of integrated, composable tooling for backup automation out of the box. On Windows, achieving the same level of automation typically requires third-party commercial software. On macOS, while the Unix underpinnings provide some of these tools, the server ecosystem is negligible. Linux stands alone as the platform where backup automation is both native and natural.

Establishing the Right Mindset

Before you proceed to the technical chapters that follow, it is essential to establish the right mindset about backups. This mindset can be summarized in a few principles that experienced Linux administrators learn, often the hard way.

Backups are not optional. They are as fundamental to system administration as user management, network configuration, and security hardening. A server without automated backups is an incomplete server, regardless of how well everything else is configured.

Untested backups are not backups. A backup that has never been restored is a hope, not a guarantee. Every backup strategy must include regular restoration testing. This means periodically restoring data from your backups to a separate location and verifying its integrity. The `borg extract` and `rsync` restoration processes should be practiced and documented before an emergency occurs.

The 3-2-1 rule is a minimum standard. The 3-2-1 backup rule states that you should maintain at least three copies of your data, on at least two different types of storage media, with at least one copy stored offsite. For critical systems, this rule should be extended to 3-2-1-1-0: three copies, two media types, one offsite, one offline or air-gapped, and zero errors in your restoration tests.

Automation is not a one-time task. Backup automation requires ongoing maintenance. As your systems change, your backup configurations must change with them. New directories are created, new databases are deployed, new services are added. Your backup scripts and configurations must evolve alongside your infrastructure. Schedule regular reviews of your backup strategy, ideally quarterly at a minimum.

Monitor everything. An automated backup that fails silently is worse than no backup at all because it creates a false sense of security. Every automated backup should produce logs, and those logs should be monitored. Failures should generate alerts. Successes should be recorded and periodically reviewed to ensure that backup sizes and durations remain within expected parameters.

Let us put this mindset into practice with a simple exercise that you can perform on any Linux system right now.

Exercise: Verify Your Current Backup Status

Open a terminal on your Linux system and run the following commands to assess your current backup situation:

```
# Check if rsync is installed
which rsync && rsync --version | head -1

# Check if Borg is installed
which borg && borg --version

# Check if any backup-related cron jobs exist
crontab -l 2>/dev/null | grep -i backup

# Check system-wide cron directories for backup scripts
ls /etc/cron.daily/ /etc/cron.weekly/ /etc/cron.monthly/ 2>/dev/
null | grep -i backup

# Check if any systemd timers related to backup exist
systemctl list-timers 2>/dev/null | grep -i backup

# Check recent backup logs if they exist
ls -la /var/log/backup* 2>/dev/null
```

If any of these commands return empty results, that tells you something important about the current state of your system's backup readiness. By the time you finish this book, every one of these commands will return meaningful results on your systems.

Note: The `which` command locates the executable file associated with a given command by searching through the directories listed in the `PATH` environment variable. If rsync or Borg is not found, they can be installed using your distribution's package manager. On Debian and Ubuntu systems, use `sudo apt install rsync borgbackup`. On Red Hat, CentOS, and Fedora systems, use `sudo dnf install rsync borgbackup`. On Arch Linux, use `sudo pacman -S rsync borg`.

What Lies Ahead

This chapter has established the foundational understanding of why backup automation matters, particularly in Linux environments. We have examined the real and present dangers of data loss, distinguished between manual backups and truly automated backup systems, explored why Linux provides the ideal platform for backup automation, and established the mindset that will guide your approach throughout the rest of this book.

In the chapters that follow, we will move from philosophy to practice. You will learn to wield rsync with precision, understanding not just its basic syntax but its advanced features for bandwidth management, partial transfers, and complex inclusion and exclusion patterns. You will master Borg's deduplication, encryption, and repository management capabilities. You will build comprehensive backup scripts in Bash that incorporate error handling, logging, notification, and retention management. You will learn to schedule these scripts using both cron and systemd timers, and you will implement monitoring to ensure that your automated backups remain healthy over time.

Every concept will be grounded in practical, real-world Linux scenarios. Every command will be explained in detail. Every script will be production-ready, not a toy example. By the end of this book, you will have the knowledge and the tools to implement robust, automated backup systems on any Linux infrastructure, from a single personal server to a fleet of hundreds of machines.

The data on your Linux systems is valuable. It deserves to be protected with the same rigor and automation that you apply to every other aspect of your infrastructure. Let us begin that work.

Chapter 2: Backup Types and Storage Models

Understanding backup strategies is one of the most critical responsibilities for any Linux system administrator. When data loss occurs, and it inevitably will, the difference between a minor inconvenience and a catastrophic disaster often comes down to the quality of your backup plan. This chapter explores the foundational concepts behind backup types and storage models as they apply to Linux environments. We will examine how full, incremental, and differential backups work, how deduplication saves storage space, and how various storage destinations from local disks to cloud endpoints fit into a comprehensive backup architecture. By the end of this chapter, you will have a thorough understanding of the theory and practice behind choosing the right backup approach for any Linux system.

Understanding Why Backup Types Matter in Linux

Before diving into the specifics of each backup type, it is important to appreciate why this knowledge matters. Linux servers often run mission-critical workloads: web applications, databases, mail servers, file shares, and virtualization platforms. Each of these workloads generates data at different rates and has different recovery requirements. A database server that processes thousands of transactions per hour has very different backup needs compared to a static web server that changes only when new content is deployed.

The choice of backup type directly affects three things: how much storage space your backups consume, how long each backup operation takes to complete, and how quickly you can restore data when disaster strikes. These three factors, storage efficiency, backup window, and recovery time, form the core triad that every backup strategy must balance.

In Linux, the tools available for implementing these backup types are both powerful and flexible. Utilities like `rsync`, `tar`, `cp`, and specialized tools like Borg Backup each handle backup types in their own way. Understanding the underlying concepts ensures you can make informed decisions regardless of which tool you ultimately choose.

Full Backups: The Foundation of Every Strategy

A full backup is exactly what the name implies: a complete copy of every file and directory within the defined backup scope. When you perform a full backup of a Linux filesystem, every single file is read, copied, and stored at the backup destination. Nothing is skipped, nothing is excluded (unless you explicitly define exclusions), and nothing is assumed from a previous backup.

The primary advantage of a full backup is simplicity during restoration. If you need to recover your entire system, you take the most recent full backup and restore it. There is no dependency on any other backup set. This independence makes full backups the most reliable and straightforward type to restore from.

However, full backups come with significant costs. Consider a Linux server with 500 gigabytes of data. If you perform a full backup every night, you consume 500 gigabytes of storage each time. Over the course of a week, that amounts to 3.5 terabytes of backup storage for a single server. The backup window is also substantial

because every file must be read and transferred, even if only a handful of files changed since the last backup.

Here is a simple example of creating a full backup using `tar` on a Linux system:

```
tar -czpf /backup/full-backup-$(date +%Y%m%d).tar.gz /home /etc /
var
```

This command creates a compressed archive of the `/home`, `/etc`, and `/var` directories. The `-p` flag preserves file permissions, which is essential for Linux backups because the permission model is integral to system security and functionality. The `$(date +%Y%m%d)` portion dynamically inserts the current date into the filename, making it easy to identify when each backup was created.

Using `rsync` for a full backup looks different but accomplishes the same goal:

```
rsync -avz --delete /home /etc /var /backup/full/
```

The `rsync` approach copies all files to the destination directory. The `-a` flag enables archive mode, which preserves permissions, ownership, timestamps, and symbolic links. The `-v` flag enables verbose output so you can monitor progress, and `-z` enables compression during transfer. The `--delete` flag ensures that files removed from the source are also removed from the backup destination, keeping the backup an accurate mirror.

Note: Full backups should always be the starting point of any backup strategy. Even strategies that rely heavily on incremental or differential backups must begin with a full backup as the baseline. Without this baseline, incremental and differential backups have no reference point.

Incremental Backups: Efficiency Through Change Tracking

An incremental backup captures only the files that have changed since the last backup of any type. This means the first incremental backup after a full backup contains only the files that changed since that full backup. The second incremental backup contains only the files that changed since the first incremental backup, and so on.

This approach dramatically reduces both storage consumption and backup time. If your 500-gigabyte server only changes 5 gigabytes of data per day, each incremental backup is approximately 5 gigabytes rather than 500 gigabytes. Over a week, instead of consuming 3.5 terabytes as a full-backup-only strategy would, you consume roughly 530 gigabytes: one 500-gigabyte full backup plus six 5-gigabyte incremental backups.

The trade-off appears during restoration. To restore a complete system from incremental backups, you must first restore the most recent full backup and then apply every incremental backup in sequence. If you performed a full backup on Sunday and it is now Friday, you would need to restore the Sunday full backup, then Monday's incremental, then Tuesday's, Wednesday's, Thursday's, and finally Friday's. If any single incremental backup in this chain is corrupted or missing, you cannot complete the restoration beyond that point.

In Linux, `rsync` can perform incremental backups using the `--link-dest` option, which creates hard links to unchanged files from a previous backup:

```
rsync -avz --link-dest=/backup/2024-01-14 /home /etc /var /
backup/2024-01-15/
```

This command compares the current state of the source directories against the previous backup at `/backup/2024-01-14`. Files that have not changed are repre-

sented as hard links rather than new copies, consuming virtually no additional disk space. Files that have changed are copied in full. The result is a backup directory that appears to contain a complete copy of the data but actually uses minimal additional storage.

Borg Backup takes this concept even further with its built-in deduplication engine, which we will explore later in this chapter.

The following table summarizes the key characteristics of incremental backups compared to full backups:

Characteristic	Full Backup	Incremental Backup
Data Captured	All files in scope	Only files changed since last backup
Storage Required	High (complete copy each time)	Low (only changes stored)
Backup Speed	Slow (all files processed)	Fast (only changed files processed)
Restore Speed	Fast (single backup needed)	Slower (full plus all incrementals needed)
Restore Complexity	Simple	Complex (chain dependency)
Risk of Data Loss	Low (self-contained)	Higher (chain must be intact)

Differential Backups: A Middle Ground

Differential backups occupy the space between full and incremental backups. A differential backup captures all files that have changed since the last full backup, regardless of any differential backups that may have occurred in between.

Consider the same 500-gigabyte server. On Sunday, you perform a full backup. On Monday, 5 gigabytes of data change, so Monday's differential backup is 5 giga-

bytes. On Tuesday, an additional 3 gigabytes change (for a total of 8 gigabytes changed since Sunday), so Tuesday's differential backup is 8 gigabytes. By Friday, the differential backup might be 25 gigabytes because it captures all changes accumulated since Sunday.

The advantage of differential backups becomes clear during restoration. To restore the system, you need only two backup sets: the most recent full backup and the most recent differential backup. This is significantly simpler and faster than the incremental approach, which requires the full backup plus every incremental in sequence.

The disadvantage is that differential backups grow larger each day as more changes accumulate. They consume more storage than incremental backups but less than performing full backups every day.

In Linux, implementing differential backups can be accomplished using `find` combined with `tar`:

```
find /home /etc /var -newer /backup/full-backup-timestamp-file
-print0 | \
    tar -czpf /backup/diff-backup-$(date +%Y%m%d).tar.gz --null
-T -
```

This command uses `find` to locate all files newer than a reference timestamp file (created at the time of the last full backup) and pipes that file list to `tar` for archiving. The `-print0` and `--null` flags handle filenames containing spaces or special characters safely, which is a common concern on Linux systems where filenames can contain virtually any character.

Another approach uses `rsync` with a carefully managed reference point:

```
rsync -avz --compare-dest=/backup/last-full/ /home /etc /var /
backup/diff-$(date +%Y%m%d)/
```

The `--compare-dest` option tells `rsync` to compare source files against the specified directory and only transfer files that differ. Unlike `--link-dest`, it does

not create hard links for unchanged files, so the resulting backup directory contains only the changed files.

Note: The choice between incremental and differential backups often depends on your recovery time objectives. If fast restoration is critical and you can afford slightly more storage, differential backups are often preferred. If storage is at a premium and you can tolerate longer restoration times, incremental backups are the better choice.

Deduplication: Eliminating Redundancy at the Block Level

Deduplication is a storage optimization technique that identifies and eliminates duplicate copies of data. Rather than storing the same data multiple times, a deduplication system stores one copy and uses references (similar in concept to hard links or pointers) wherever that data appears again.

There are two primary levels of deduplication: file-level and block-level. File-level deduplication identifies identical files and stores only one copy. Block-level deduplication goes further by breaking files into smaller chunks (blocks) and identifying duplicate blocks across all files. This means that even if two files are not identical, any shared portions between them are stored only once.

Borg Backup implements content-defined chunking, a sophisticated form of block-level deduplication. When Borg processes a file, it divides the file into variable-length chunks based on the content itself rather than fixed-size blocks. This approach is resilient to insertions and deletions within files because shifting content does not cause all subsequent chunk boundaries to change, as would happen with fixed-size chunking.

Here is an example of creating a Borg repository and performing an initial backup:

```
borg init --encryption=repokey /backup/borg-repo

borg create /backup/borg-repo::full-{now:%Y-%m-%d} /home /etc /
var
```

The first command initializes a new Borg repository with encryption. The second command creates an archive within that repository. When subsequent backups are created:

```
borg create /backup/borg-repo::daily-{now:%Y-%m-%d} /home /etc /
var
```

Borg automatically deduplicates the data. If 95 percent of the data is identical to the previous backup, only the 5 percent that changed is actually stored. The archive appears to contain a complete copy of all the data, but the actual storage consumed is minimal.

You can verify the deduplication efficiency using:

```
borg info /backup/borg-repo
```

This command displays statistics including the original size of all archives, the deduplicated size, and the compression ratio. It is common to see deduplication ratios of 10:1 or higher in environments where backups are performed daily and data change rates are modest.

The following table compares deduplication approaches:

Deduplication Type	Granularity	Storage Savings	Processing Overhead	Example Tools
No Deduplication	None	None	Minimal	tar, cp