PowerShell 7.x and SQLite Fundamentals

Lightweight Data Storage and Automation with PowerShell and SQLite

Preface

Every PowerShell scripter eventually hits the same wall. You start with simple scripts, graduate to managing configuration in CSV files and JSON, and then one day you realize your automation needs have outgrown flat files. You need something more structured, more reliable, and more queryable—but standing up a full SQL Server instance feels like bringing a sledgehammer to hang a picture frame.

That's exactly where SQLite changes everything.

PowerShell 7.x and SQLite Fundamentals was written for the PowerShell practitioner who needs a lightweight, zero-configuration data storage solution that fits naturally into the scripting and automation workflows they already know. SQLite is the most widely deployed database engine in the world, and yet it remains surprisingly underutilized in the PowerShell ecosystem. This book aims to change that.

What This Book Is About

At its core, this is a PowerShell book. Every concept, every technique, and every example is framed through the lens of PowerShell 7.x. You won't find abstract database theory here—you'll find practical, hands-on guidance for integrating SQLite into your PowerShell scripts, tools, and automation pipelines.

We begin by establishing *why* PowerShell and SQLite are such a natural pairing and then walk through the essential SQLite concepts that matter most to PowerShell users. From there, we move into the practical work: setting up your PowerShell environment, creating databases, writing SQL queries, and—critically—working with query results as the rich PowerShell objects you're accustomed to.

The middle chapters are where things get particularly exciting. You'll learn how to use SQLite to persist script state across sessions, power scheduled automation tasks, handle transactions and error recovery gracefully, and secure your data stores. These are the real-world challenges that PowerShell professionals face daily, and SQLite provides elegant solutions to all of them.

The final chapters bring everything together. You'll build a complete data-driven PowerShell tool, implement inventory and reporting solutions, and internalize the best practices that separate fragile scripts from production-ready automation. The book closes by pointing you toward advanced data automation patterns, ensuring this is a beginning rather than an end.

Who This Book Is For

This book is for **PowerShell scripters, system administrators, DevOps engineers, and automation professionals** who want to add structured data storage to their toolkit without the overhead of a client-server database. If you're comfortable writing PowerShell scripts and want to level up your data management capabilities, you're in the right place. No prior database experience is required—Chapter 5 covers exactly the SQL you need, and nothing you don't.

How This Book Is Structured

The sixteen chapters follow a deliberate progression from foundations to real-world application. Chapters 1–3 set the stage. Chapters 4–8 build your core skills. Chapters 9–12 tackle intermediate patterns like state management, scheduling, security, and error handling. Chapters 13–16 culminate in applied projects and forward-looking guidance. The five appendices serve as ongoing references you'll return to long after your first read—including a SQL cheat sheet, reusable helper functions, troubleshooting guides, and complete example projects.

Acknowledgments

No technical book is a solo endeavor. I owe deep gratitude to the PowerShell community—a remarkably generous group of professionals who share knowledge freely and lift each other up. The creators and maintainers of SQLite deserve recognition for building one of the most reliable pieces of software ever written. Thanks also to the technical reviewers whose sharp eyes and honest feedback made this a better book, and to every reader who has ever asked, *"Is there something simpler than SQL Server for my PowerShell scripts?"* This book is the answer.

Whether you're tracking server inventory, logging automation results, managing configuration data, or building lightweight tools for your team, the combination of **PowerShell 7.x and SQLite** will give you capabilities that are powerful, portable, and remarkably simple. Let's get started.

Author: Laszlo Bocso (MCT)

Table of Contents

Chapter 1: Why PowerShell and SQLite Belong Together

Introduction

There is a quiet revolution happening in the world of system administration and automation. For years, administrators and developers working in the PowerShell ecosystem have relied on flat files, CSV exports, XML documents, and JSON structures to store and retrieve data. While these approaches have served their purpose, they carry inherent limitations that become painfully obvious as projects grow in complexity. Enter SQLite, a lightweight, serverless, self-contained relational database engine that fits into the PowerShell workflow as naturally as a cmdlet fits into a pipeline. This chapter explores why the combination of PowerShell and SQLite is not merely convenient but genuinely powerful, and why understanding this pairing will elevate your scripting and automation capabilities to a professional level.

Before we dive into the specifics, let us establish a shared understanding. PowerShell 7.x, the cross-platform evolution of Windows PowerShell, is a task automation framework built on the .NET runtime. It provides a command-line shell, a scripting language, and a configuration management framework. SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured SQL relational database engine. Together, they form a partnership that addresses a gap many PowerShell practitioners have felt but may not have articulated: the need for structured, queryable, persistent data storage without the overhead of a full database server.

1.1 The Evolution of PowerShell as a Data-Driven Tool

PowerShell was born out of necessity. In the early 2000s, Microsoft recognized that its command-line tools were falling behind the Unix world. Jeffrey Snover's Monad Manifesto laid the groundwork for what would become PowerShell, a shell that treated everything as an object rather than a stream of text. This object-oriented approach was revolutionary. When you run `Get-Process` in PowerShell, you do not receive lines of text that you must parse with regular expressions. You receive rich .NET objects with properties and methods that you can inspect, filter, sort, and manipulate directly.

Over the years, PowerShell has grown from a Windows-only administrative tool into a cross-platform powerhouse. PowerShell 7.x runs on Windows, macOS, and Linux. It integrates with Azure, AWS, and Google Cloud. It manages Active Directory, Exchange, SharePoint, and hundreds of other services through modules. But throughout this evolution, one theme has remained constant: PowerShell is fundamentally a data-driven tool.

Consider the typical workflow of a PowerShell script. You gather data from one or more sources, transform that data, make decisions based on it, and then output results or take actions. This is the Extract-Transform-Load pattern that database professionals have used for decades. PowerShell excels at the extraction and transformation phases. Where it has traditionally been weaker is in the persistent storage phase, the part where you need to save data, query it later, relate it to other data, and maintain its integrity over time.

This is where most PowerShell practitioners reach for CSV files, and this is where the limitations begin to surface.

1.2 The Limitations of Flat File Storage in Automation

Let us examine a common scenario. You are responsible for monitoring server health across fifty machines. You write a PowerShell script that collects CPU usage, memory consumption, disk space, and service status from each server. You need to store this data so you can track trends over time, generate reports, and trigger alerts when thresholds are exceeded.

The instinctive approach is to export the data to a CSV file using `Export-Csv`. This works beautifully for the first day. By the end of the first week, you have seven CSV files. By the end of the first month, you have thirty. Now you need to answer a question: "Which servers had CPU usage above 90 percent more than three times in the last two weeks?" Suddenly, you are writing PowerShell code to import multiple CSV files, parse date strings, filter records, group results, and count occurrences. The code becomes complex, brittle, and slow.

The following table summarizes the key limitations of common flat file formats when used for persistent data storage in PowerShell automation scenarios.

Storage Format	Limitation	Impact on PowerShell Workflows
CSV	No data types; everything is a string	Requires manual type conversion with `[int]`, `[datetime]`, and similar casts every time data is imported

CSV	No relational capability	Cannot join data from different collections without writing custom PowerShell logic
CSV	No concurrent access safety	Multiple scripts writing to the same file can cause corruption or data loss
CSV	No indexing	Searching large files requires reading the entire file into memory with `Import-Csv`
JSON	Nested structures become unwieldy	Deep property access chains like `$data.servers.network.interfaces.ipv4` are fragile and hard to maintain
JSON	No built-in query language	Filtering requires loading the entire document and using `Where-Object`
XML	Verbose syntax increases file size	Large datasets produce enormous files that are slow to parse with `[xml]` or `Select-Xml`
XML	Complex manipulation	Adding, updating, and deleting nodes requires understanding the XML DOM, which adds cognitive overhead
Registry	Windows-only	Not available on macOS or Linux, breaking cross-platform compatibility in PowerShell 7.x
Plain Text	No structure whatsoever	Requires custom parsing logic with `-match`, `-replace`, or `[regex]` for every read operation

These limitations do not mean that CSV, JSON, and XML are bad formats. They are excellent for data interchange, configuration, and simple logging. But they are not databases, and when you try to use them as databases, you inherit all the problems that databases were invented to solve.

1.3 What SQLite Brings to the Table

SQLite is not a new technology. It was created by D. Richard Hipp in the year 2000 and has since become the most widely deployed database engine in the world. It runs inside every Android phone, every iPhone, every Mac, every Windows 10 and later machine, every major web browser, and countless embedded devices. Its reliability is legendary; the SQLite development team maintains 100 percent branch test coverage with millions of test cases.

What makes SQLite uniquely suited to PowerShell work is its architecture. Unlike SQL Server, PostgreSQL, or MySQL, SQLite does not require a server process. There is no service to install, no port to configure, no authentication to manage, and no daemon to monitor. A SQLite database is a single file on disk. You can create it, query it, back it up by copying the file, and delete it when you are done. This simplicity aligns perfectly with the PowerShell philosophy of getting things done without unnecessary ceremony.

Here are the characteristics that make SQLite an ideal companion for PowerShell.

SQLite Characteristic	**Benefit for PowerShell Users**
Serverless architecture	No installation or configuration required; just load the assembly and start working
Single-file database	Easy to create, copy, move, and back up using standard PowerShell file commands like `Copy-Item`
Zero configuration	No connection strings with server names, ports, or credentials for local use
ACID compliant	Guarantees that your data remains consistent even if a script terminates unexpectedly
Cross-platform	Works identically on Windows, macOS, and Linux, matching PowerShell 7.x's cross-platform nature

Full SQL support	Provides SELECT, INSERT, UPDATE, DELETE, JOIN, GROUP BY, subqueries, views, triggers, and more
Small footprint	The entire engine is roughly 1 MB, trivial compared to any server-based database
Public domain	No licensing concerns, no cost, no restrictions on use in commercial or government environments
Type affinity system	Supports INTEGER, REAL, TEXT, BLOB, and NULL, providing meaningful data types without rigid enforcement
Built-in functions	Offers date and time functions, string functions, aggregate functions, and mathematical functions

When you combine these characteristics with PowerShell's pipeline, object model, and scripting capabilities, you get a data management solution that is more powerful than flat files but simpler than a full database server. You get the ability to write SQL queries against your automation data, to create indexes that make searches fast, to define relationships between tables, and to ensure data integrity through constraints and transactions.

1.4 Real-World Use Cases for the PowerShell and SQLite Combination

To appreciate why PowerShell and SQLite belong together, let us examine several real-world scenarios where this combination shines.

Inventory Management. System administrators often need to maintain an inventory of hardware, software, and configurations across their environment. A SQLite database can store server names, IP addresses, operating system versions, installed software, hardware specifications, and configuration settings in properly normalized tables. PowerShell scripts can update this inventory on a schedule using `Invoke-Command` to gather data remotely and SQL INSERT or UPDATE statements to persist it. When you need to find all servers running a specific version of .NET, a single SQL query returns the answer in milliseconds, regardless of whether your inventory contains fifty or fifty thousand records.

Log Aggregation and Analysis. PowerShell scripts that monitor event logs, application logs, or custom log sources can write parsed log entries into a SQLite database. Because SQLite supports indexes, you can create indexes on timestamp columns, severity levels, source names, or any other field you frequently query. This transforms log analysis from a tedious exercise in file parsing into a straightforward SQL query.

Configuration Drift Detection. You can capture the desired state of your systems in a SQLite database and then write PowerShell scripts that compare the current state against the desired state. The relational nature of SQLite allows you to model complex configurations with multiple related tables, something that would be extremely awkward with flat files.

Report Generation. PowerShell scripts that generate reports for management or compliance purposes benefit enormously from SQLite's aggregation functions. Instead of writing complex PowerShell grouping and calculation logic, you can write a SQL query with GROUP BY, COUNT, SUM, AVG, and other aggregate functions, then format the results using PowerShell's `Format-Table` or export them with `Export-Csv`.

Script State Persistence. Long-running or scheduled PowerShell scripts often need to remember their state between executions. Which items have already been processed? When was the last successful run? What was the last error? A SQLite database provides a robust, queryable mechanism for persisting this state information.

Testing and Development. When developing PowerShell modules or functions that interact with databases, SQLite provides an excellent testing backend. You can create an in-memory SQLite database that exists only for the duration of your test, populate it with test data, run your code against it, and let it disappear when the test completes. This eliminates the need for a test database server.

1.5 Comparing Approaches: Before and After SQLite

To make the contrast concrete, consider a simple example. You want to store a list of servers with their roles and last patch dates, then find all web servers that have not been patched in the last 30 days.

The CSV Approach in PowerShell:

```
# Writing data
$servers = @(
    [PSCustomObject]@{Name='SRV01'; Role='Web';
LastPatched='2024-11-15'}
    [PSCustomObject]@{Name='SRV02'; Role='Database';
LastPatched='2024-12-01'}
    [PSCustomObject]@{Name='SRV03'; Role='Web';
LastPatched='2024-10-20'}
    [PSCustomObject]@{Name='SRV04'; Role='Web';
LastPatched='2024-12-10'}
)
$servers | Export-Csv -Path "C:\Data\servers.csv"
-NoTypeInformation

# Reading and querying data
$cutoffDate = (Get-Date).AddDays(-30)
$results = Import-Csv -Path "C:\Data\servers.csv" |
    Where-Object {
        $_.Role -eq 'Web' -and
        [datetime]$_.LastPatched -lt $cutoffDate
    }
$results
```

Notice that you must manually cast the `LastPatched` string to a `[datetime]` object because CSV files have no concept of data types. If someone edits the CSV file and introduces an invalid date format, your script will throw a runtime error. If the CSV file grows to contain thousands of records, every query requires reading the entire file into memory.

The SQLite Approach in PowerShell:

```
# Establishing a connection and creating the table
$connectionString = "Data Source=C:\Data\servers.db"
$connection = New-Object
System.Data.SQLite.SQLiteConnection($connectionString)
$connection.Open()

$createTable = $connection.CreateCommand()
$createTable.CommandText = @"
    CREATE TABLE IF NOT EXISTS Servers (
        Name TEXT PRIMARY KEY,
        Role TEXT NOT NULL,
        LastPatched TEXT NOT NULL
    )
"@
$createTable.ExecuteNonQuery()

# Inserting data
$insertCmd = $connection.CreateCommand()
$insertCmd.CommandText = "INSERT OR REPLACE INTO Servers (Name,
Role, LastPatched) VALUES (@Name, @Role, @LastPatched)"
$insertCmd.Parameters.AddWithValue("@Name", "SRV01")
$insertCmd.Parameters.AddWithValue("@Role", "Web")
$insertCmd.Parameters.AddWithValue("@LastPatched", "2024-11-15")
$insertCmd.ExecuteNonQuery()

# Querying data
$queryCmd = $connection.CreateCommand()
$queryCmd.CommandText = "SELECT * FROM Servers WHERE Role = 'Web'
AND LastPatched < date('now', '-30 days')"
$reader = $queryCmd.ExecuteReader()

while ($reader.Read()) {
    Write-Output "Server: $($reader['Name']), Last Patched: $
($reader['LastPatched'])"
}

$connection.Close()
```

The SQLite approach requires more initial setup, but the query itself is a single SQL statement that handles date comparison natively, runs against an indexed data store, and does not require loading the entire dataset into memory. As your data grows, the SQL approach scales gracefully while the CSV approach degrades.

Note: The code examples above use the `System.Data.SQLite` .NET assembly. In later chapters, we will explore how to install this assembly, how to use the `PSSQLite` module that wraps these operations in PowerShell-friendly cmdlets, and how to build your own helper functions. For now, the purpose is to illustrate the conceptual difference between flat file and database approaches.

1.6 Setting Expectations for This Book

This book is designed for PowerShell practitioners who want to add structured data management to their toolkit without the overhead of learning database administration. You do not need prior experience with SQL or databases. Each chapter builds on the previous one, starting with installation and basic operations, progressing through advanced queries and data modeling, and culminating in real-world projects that demonstrate the full power of the PowerShell and SQLite combination.

Throughout this book, every concept is presented through the lens of PowerShell. When we discuss SQL syntax, we show how to execute it from PowerShell. When we discuss database design, we frame it in terms of the PowerShell objects and data structures you already understand. When we discuss performance optimization, we measure it using PowerShell's `Measure-Command` cmdlet.

The following table outlines what you can expect to learn as you progress through the chapters.

Topic Area	**What You Will Learn**	**PowerShell Skills Applied**
Installation and Setup	How to obtain and configure SQLite for use with PowerShell 7.x on any platform	Module installation with `Install-Module`, assembly loading with `Add-Type`
Basic Operations	Creating databases, tables, inserting data, and running queries	Working with .NET objects, string formatting, pipeline output
Data Modeling	Designing tables and relationships for real automation scenarios	Understanding PSCustomObject structures and how they map to database rows
Advanced Queries	Joins, subqueries, aggregations, and window functions	Comparing SQL operations to equivalent PowerShell pipeline operations
Transactions and Error Handling	Ensuring data integrity and handling failures gracefully	Try-Catch-Finally blocks, ErrorAction preferences, transaction management
Performance Optimization	Indexing strategies, bulk operations, and query planning	Measure-Command, profiling, batch processing techniques
Real-World Projects	Complete solutions for inventory, logging, reporting, and more	Combining all skills into production-ready PowerShell scripts and modules

Summary

PowerShell and SQLite belong together because they share a common philosophy: provide maximum capability with minimum overhead. PowerShell gives you the ability to automate, transform, and orchestrate. SQLite gives you the ability to persist, query, and protect your data. Together, they fill a gap that flat files cannot address and server-based databases make unnecessarily complex.

As you move through this book, you will discover that adding SQLite to your PowerShell workflow is not a dramatic shift in how you work. It is a natural extension of what you already do. You will still write PowerShell scripts. You will still use the pipeline. You will still work with objects. But you will have a powerful, reliable, standards-based data store behind your scripts, one that scales from a simple configuration file replacement to a full-featured local database supporting complex queries and relationships.

The journey begins in the next chapter, where we will install everything you need and create your first SQLite database from the PowerShell command line.

Chapter 2: SQLite Concepts for PowerShell Users

Introduction

Before you can effectively harness the power of SQLite within your PowerShell scripts and automation workflows, you need to understand the fundamental concepts that make SQLite unique among database systems. This chapter bridges the gap between traditional database knowledge and the practical reality of working with SQLite through PowerShell. Whether you are a systems administrator who has never touched a database or a developer transitioning from SQL Server, this chapter will ground you in the essential concepts you need to write effective PowerShell scripts that interact with SQLite databases.

SQLite is not merely a smaller version of enterprise databases like SQL Server or PostgreSQL. It operates on a fundamentally different philosophy, and understanding that philosophy will directly influence how you design your PowerShell solutions. Throughout this chapter, we will examine SQLite's architecture, its type system, its approach to data storage, and how each of these concepts maps directly to the PowerShell environment you already know.

2.1 What is SQLite and Why Use It with PowerShell

SQLite is a self-contained, serverless, zero-configuration, transactional SQL database engine. Unlike SQL Server, MySQL, or PostgreSQL, SQLite does not require a separate server process running in the background. The entire database is stored in a single file on disk, and your PowerShell script communicates with it directly through a library rather than over a network connection.

This architecture makes SQLite an extraordinarily powerful companion for PowerShell scripting. Consider the scenarios where you currently use CSV files, XML documents, or JSON files to store structured data. SQLite offers all the querying power of SQL while maintaining the simplicity of a single file that you can copy, move, back up, or email just like any other file.

The following table outlines the key characteristics of SQLite and how they benefit PowerShell users:

Characteristic	Description	Benefit for PowerShell Users
Serverless	No separate database server process is required	No installation overhead; scripts can run on any machine without database server setup
Self-Contained	The entire database engine is a single library	Easy to distribute with PowerShell modules; no external dependencies to manage
Zero-Configuration	No setup, administration, or tuning required	PowerShell scripts can create and use databases immediately without DBA involvement
Single File	The entire database is stored in one cross-platform file	Easy to back up, copy, or transfer using standard PowerShell file commands like Copy-Item

Transactional	Full ACID compliance for data integrity	Safe concurrent access from multiple PowerShell runspaces or scheduled tasks
Cross-Platform	Works on Windows, Linux, and macOS	Perfect match for PowerShell 7.x, which also runs on all three platforms

When you compare this to the approach many PowerShell scripters take with flat files, the advantages become clear. A CSV file has no built-in indexing, no query language beyond what PowerShell provides through `Where-Object` and `Sort-Object`, and no transactional safety if your script crashes mid-write. SQLite gives you all of these features while remaining almost as simple to work with as a flat file.

Note: SQLite is not a replacement for enterprise database systems. If you need multi-user concurrent write access from dozens of clients, network-accessible database services, or the advanced features of SQL Server, those tools remain the right choice. SQLite excels in scenarios where PowerShell scripts need local, structured data storage with powerful querying capabilities.

2.2 Understanding the SQLite Architecture

To write effective PowerShell scripts that interact with SQLite, you need to understand how SQLite organizes and manages data internally. This understanding will help you make better decisions about database design and troubleshoot issues when they arise.

The Database File

Every SQLite database is a single ordinary file on the filesystem. When you create a database in your PowerShell script, you are creating a file. When you query a database, you are reading a file. This is a profound simplification compared to client-server database systems.

The database file has a well-defined format that begins with a 100-byte header. This header contains metadata about the database, including the page size, file format versions, and the size of the database in pages. The rest of the file is divided into fixed-size pages, typically 4096 bytes each.

```
# Creating a SQLite database is as simple as specifying a file
path
$databasePath = "C:\Data\inventory.db"

# When you open a connection to a non-existent file, SQLite
creates it
# This is equivalent to creating a new, empty database
$connectionString = "Data Source=$databasePath;Version=3;"
```

Pages and the B-Tree Structure

SQLite organizes data within the database file using a B-tree data structure. There are two types of B-trees used internally:

B-Tree Type	Purpose	PowerShell Relevance
Table B-Tree	Stores actual table data with integer rowid as the key	Every table you create from PowerShell uses one of these
Index B-Tree	Stores index entries for fast lookups	Created when you add indexes to improve query performance in your scripts

Each page in the database file belongs to one of these B-trees, or it serves as an overflow page for large data, or it is a free page that was previously used but is now available for reuse.

Understanding this structure matters for PowerShell users because it explains why certain operations are fast and others are slow. When your PowerShell script queries a table by its rowid or primary key, SQLite can navigate the B-tree efficiently. When your script performs a full table scan with a complex `WHERE` clause on an unindexed column, SQLite must read every page of the table's B-tree.

The Journal and Write-Ahead Log

When your PowerShell script modifies data in a SQLite database, the changes do not go directly into the main database file. SQLite uses one of two mechanisms to ensure data integrity:

Rollback Journal Mode: Before modifying a page, SQLite copies the original page content to a separate journal file. If the operation fails or your PowerShell script crashes, SQLite can restore the original pages from the journal.

Write-Ahead Log (WAL) Mode: Instead of writing changes to the main database file, SQLite appends them to a separate WAL file. Readers can continue reading the main database file while a writer appends to the WAL. This mode generally provides better concurrency, which matters when multiple PowerShell processes might access the same database.

```
# You can set the journal mode from PowerShell when configuring
your database
# WAL mode is often preferred for better concurrent access
$pragmaCommand = "PRAGMA journal_mode=WAL;"
```

Note: The WAL mode is particularly valuable when you have PowerShell scheduled tasks that might overlap in execution, or when you are running background jobs that access the same SQLite database as your interactive session.

2.3 SQLite Data Types and PowerShell Type Mapping

One of the most distinctive features of SQLite is its type system, and understanding it is critical for PowerShell users who are accustomed to the strongly-typed world of .NET objects.

SQLite Type Affinity

Unlike SQL Server, where a column declared as `INTEGER` will reject any attempt to store a string, SQLite uses a concept called "type affinity." The declared type of a column is a recommendation, not a strict requirement. SQLite can store any type of value in any column, regardless of the declared type.

SQLite has five storage classes:

Storage Class	Description	PowerShell .NET Equivalent
NULL	Represents a missing or unknown value	`$null`
INTEGER	A signed integer stored in 1, 2, 3, 4, 6, or 8 bytes	`[System.Int64]` or `[long]`
REAL	A floating-point number stored as an 8-byte IEEE float	`[System.Double]` or `[double]`

TEXT	A text string stored using the database encoding (UTF-8, UTF-16)	`[System.String]` or `[string]`
BLOB	Binary data stored exactly as input	`[System.Byte[]]` or `[byte[]]`

When you declare a column type in your `CREATE TABLE` statement, SQLite assigns one of five type affinities to that column: TEXT, NUMERIC, INTEGER, REAL, or BLOB. The affinity determines how SQLite attempts to convert values stored in that column.

The following table shows how common SQL type declarations map to SQLite affinities:

Declared Column Type	**Resulting Affinity**	**Behavior**
INT, INTEGER, BIGINT, SMALLINT	INTEGER	Values are stored as integers when possible
CHAR, VARCHAR, TEXT, CLOB	TEXT	Values are stored as text strings
REAL, FLOAT, DOUBLE	REAL	Values are stored as floating-point numbers
BLOB, no type specified	BLOB (NONE)	Values are stored exactly as provided
NUMERIC, DECIMAL, BOOLEAN, DATE, DATETIME	NUMERIC	Values are stored as integer or real if possible, otherwise as text

Mapping Between PowerShell and SQLite Types

When you send data from PowerShell to SQLite or retrieve data from SQLite into PowerShell, type conversion happens at the boundary. Understanding this conversion is essential for writing reliable scripts.

```
# PowerShell variables and their SQLite storage behavior
$integerValue = 42                # Stored as INTEGER in SQLite
$floatValue = 3.14159             # Stored as REAL in SQLite
$stringValue = "Hello World"      # Stored as TEXT in SQLite
$nullValue = $null                # Stored as NULL in SQLite
$binaryValue = [byte[]](0x48, 0x65, 0x6C, 0x6C, 0x6F)  # Stored
as BLOB in SQLite
```

When retrieving data from SQLite back into PowerShell, the reverse mapping occurs. INTEGER values become `[long]` objects, REAL values become `[double]` objects, TEXT values become `[string]` objects, and BLOB values become `[byte[]]` arrays.

Note: One common pitfall for PowerShell users is the handling of dates. SQLite has no native DATE or DATETIME storage class. Dates are typically stored as TEXT in ISO 8601 format (such as "2024-01-15 14:30:00"), as INTEGER representing Unix timestamps, or as REAL representing Julian day numbers. When working with dates in PowerShell, you must explicitly convert `[datetime]` objects to one of these formats before storing them, and convert them back when retrieving.

```
# Storing a date as ISO 8601 text
$currentDate = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
# $currentDate is now a string like "2024-01-15 14:30:00"

# Storing a date as Unix timestamp
$unixTimestamp = [long](Get-Date -UFormat %s)
# $unixTimestamp is now an integer like 1705312200

# Retrieving and converting back to PowerShell datetime
$retrievedDateString = "2024-01-15 14:30:00"
$dateTimeObject = [datetime]::ParseExact($retrievedDateString,
"yyyy-MM-dd HH:mm:ss", $null)
```