Realtime publishers

The Definitive Guide™ To

Windows Application and Server Backup 2.0

sponsored by

AppAssure
HOME OF BACKUP 2.0

Don Jones

Copyright Statement

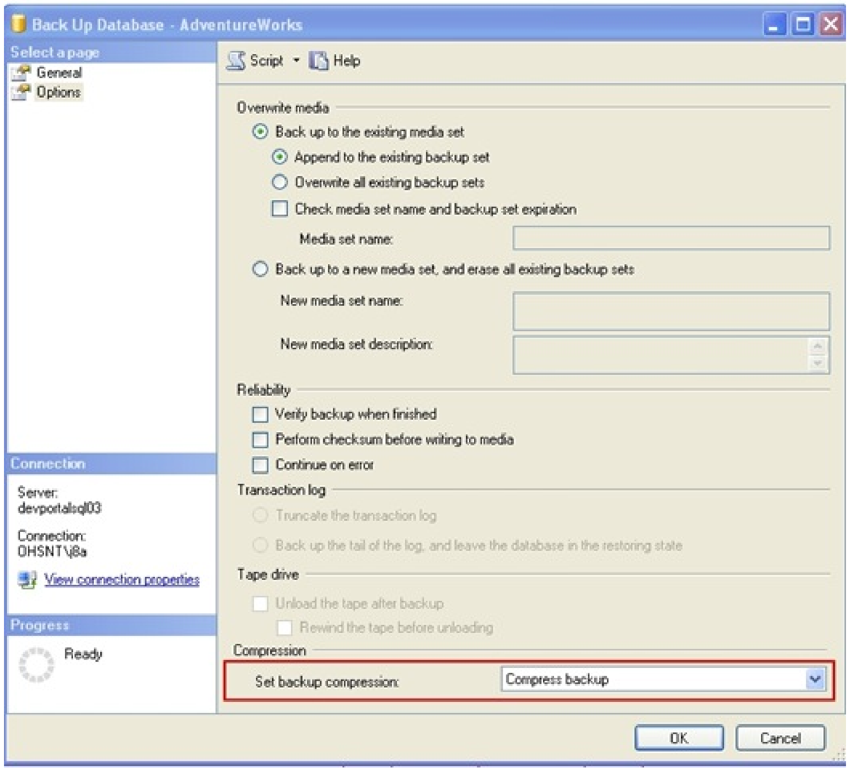
Chapter 5: SQL Server Backups

More and more companies are using Microsoft SQL Server these days—and in many cases, *they don't even realize it*. While plenty of organizations deliberately install SQL Server, many businesses find themselves using SQL Server as a side effect, because SQL Server is the data store for some line-of-business application, technology solution, and so on. In fact, "SQL sprawl" makes SQL Server one of the most challenging server products from a backup perspective: Not only is SQL Server challenging in and of itself, but you wind up with tons of instances!

Here's what I see happening in many organizations: The company has one or more "official" SQL Server installations, and the IT team is aware of the need to back up these instances on a regular basis. But there are also numerous "stealth" installations of SQL Server, often running on the "Express" edition of SQL Server, that the IT team is unaware of. The data stored in these "stealth" installations is no less mission critical than the data in the "official" installations, but in many cases, that data isn't being protected properly. Dealing with this "sprawl" is just one of the unique challenges that Backup 2.0 faces in SQL Server.

Native Solutions

SQL Server has always offered a native application programming interface (API) for backing up databases. In fact, SQL Server has long been one of the few Microsoft server applications that *natively* supports tape backup, without using Windows' own backup utility. The native backup toolset is actually quite robust, supporting features like compression (highlighted in Figure 5.1), encryption, and so forth.



Server: devportalsql03

Connection: OHSNT\j8a

View connection properties

Ready

Overwrite media

- (●) Back up to the existing media set
  - (●) Append to the existing backup set
  - ( ) Overwrite all existing backup sets
  - ☐ Check media set name and backup set expiration
    - Media set name:
- ( ) Back up to a new media set, and erase all existing backup sets
  - New media set name:
  - New media set description:

Reliability

- ☐ Verify backup when finished
- ☐ Perform checksum before writing to media
- ☐ Continue on error

Transaction log

- ( ) Truncate the transaction log
- ( ) Back up the tail of the log, and leave the database in the restoring state

Tape drive

- ☐ Unload the tape after backup
  - ☐ Rewind the tape before unloading

Compression

Set backup compression: Compress backup

Figure 5.1: SQL Server's native backup interface.

To understand SQL Server's native backup technology, you need to first know a bit about how SQL Server works under the hood.

How SQL Server Works

SQL Server stores things on disk in 8KB chunks called pages. It also manipulates those same 8KB chunks in memory, meaning the smallest unit of data SQL Server works with is 8KB.

When data is written to disk, an entire row of data must fit within that 8KB page. It's possible for multiple rows to share a page, but a row cannot span multiple pages. So, if a Customers table has columns for Name, Address, City, State, and Phone, then all that data combined must be less than 8KB. An exception is made for certain data types—such as binary data like photos, or large gobs of text—where the actual page only contains a pointer to the real data. The real data can then be spread across multiple pages, or even stored in a file. SQL Server gathers all these 8KB pages into a simple file on disk, which usually has either an .MDF or an .NDF filename extension.

When SQL Server is told to do something, it's by means of a query, written in the Structured Query Language (SQL) syntax. In the case of a "modification" query, SQL Server modifies the pages of data in memory. But it doesn't write those modifications back out to disk yet, as there might be additional changes coming along for those pages and the system load might not offer a good disk-writing opportunity right then. What SQL Server does do, however, is make a copy of the modification query in a special log file called the transaction log. This file, which has an .LDF filename extension, keeps a record of every transaction SQL Server has executed.

Eventually—maybe a few seconds later—SQL Server will decide to write the modified pages out to disk. When it does so, it goes back to the transaction log and "checks off" the transaction that made the modifications—essentially saying, "Okay, I made that change and it's been written to disk." That way SQL Server knows that the change is safe on disk.

In the event that SQL Server crashes, it has an automated recovery mode that kicks in when it starts back up. It goes straight to the transaction log and looks for uncommitted transactions—those that have not yet been "checked off." It knows that the "checked off" transactions are safe on disk; anything else had not been written to disk and was still floating around in memory when the server crashed. So SQL Server reads those transactions out of the log, re-executes them, and immediately writes the affected pages to disk. This process allows SQL Server to "catch up" with any in-progress work, and ensures that you never lose any data—provided your disk files are okay, of course.

Think about this important fact: EVERYTHING that happens in SQL Server happens only through the transaction log, and SQL Server can re-read the log to repeat whatever has happened. This process makes nearly everything that SQL Server does possible.

How SQL Server Native Backup Works

SQL Server's native backup system works in conjunction with the transaction log. Essentially, there are two types of backup SQL Server can make: data backups and log backups. Data backups are, as you might suspect, of the database itself. These are done in a Backup 1.0-style manner, grabbing a snapshot of the data as it sits during the backup. Log backups grab the contents of the transaction log.

SQL Server's native backup capabilities include the ability to back up a database while it's in use, although database performance can slow slightly while a backup operation is underway. The ability to back up an in-use database means that SQL Server is less impacted by "backup windows" than many other server products, and it means that you're a bit less tied to the Backup 1.0-model of only grabbing backups while the data isn't being used.

But that doesn't mean SQL Server is entirely free of backup problems and challenges.

Problems and Challenges

There are a few distinct challenges presented by traditional SQL Server backup techniques:

- **Sprawl.** As I mentioned earlier, most organizations have a lot more SQL Server installations than they often realize, and backing up them all can be painful. In some cases, particularly with the "Express" editions often embedded into line-of-business applications and IT tools, SQL Server is running on a client computer that isn't being treated like a server in terms of backup and recovery.
- **Snapshots.** Just like any Backup 1.0 scenario, SQL Server backups are built around the idea of point-in-time snapshots. As I'll describe in a bit, SQL Server does offer some unique abilities that let you take more snapshots more frequently, but you'll always have a certain amount of data at risk.
- **Recovery times.** Although SQL Server can be pretty flexible in how it makes you do backups, restoring is still a time-consuming operation. So time consuming, in fact, that some companies have created tools that can "attach" a database backup to SQL Server, allowing the backup data to be queried without actually having to restore the database. This trick is useful for things like change control, but it doesn't help from a backup and recovery perspective simply because the attached backup is read-only.
- **Transaction logs.** In SQL Server, backups are intimately tied to the transaction log, and backups are *required* in order to keep the transaction log from growing larger and larger. Any backup plan that doesn't use the native APIs needs to deal with this fact.

Any proposed backup solution that does *not* use SQL Server's native APIs will be challenging. In fact, most third-party backup solutions are simply agents that sit on top of SQL Server's native APIs! This setup ensures that SQL Server's internal needs—like the transaction log—are taken care of, but it also has historically limited third-party solutions to the same basic feature set as SQL Server's native capabilities. Most third-party SQL Server backup solutions are really little more than an agent that takes data from SQL Server's native APIs, and transmits that data across the network.

In the Old Days

So how has SQL Server traditionally been included in a backup and recovery plan? Let's consider some of the techniques, scenarios, and tools that are common in the Backup 1.0 world.

Backup Techniques

SQL Server natively offers three types of backup. I know I said *two* earlier, but hear me out:

- **Full Backup.** This is a complete backup of the entire database. Once made, committed transactions in the transaction log are cleared, a process called log truncation; this is what keeps transaction logs from growing forever.
- **Differential Backup.** This is also a backup of the database, but only the data that changes since the last full backup is included. Again, the transaction log is truncated.
- **Transaction Log Backup.** This doesn't grab any of the actual data; it simply grabs the current state of the transaction log—and then truncates that log.

So two kinds of *data* backup and a *log* backup. Although SQL Server *can* back up an active database, it's not something you'd do during peak database usage due to performance concerns, so full and even differential backups are still usually done during off-peak periods or during an evening or weekend maintenance window. Because it can be difficult to get a nightly full backup of large databases in that window, administrators typically resort to a tiered backup plan—grabbing full backups on the weekends, for example, and differentials each evening. To help reduce the amount of at-risk data, transaction log backups can be made periodically throughout the day. These backups are very fast and offer little performance impact, so a practical backup plan might look something like the one in Figure 5.2.

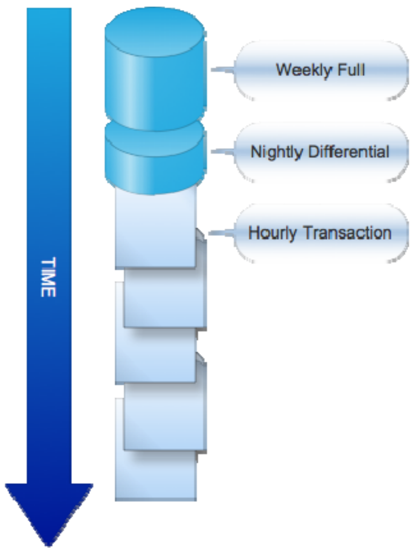


Figure 5.2: Typical SQL Server backup plan.

With this plan, the maximum amount of at-risk data is about an hour, as that's the interval between transaction log backups. Of course, in a busy database, an hour can be a *lot* of data! Reviewing our manifesto for Backup 2.0:

> Backups should prevent us from losing any data or losing any work, and ensure that we always have access to our data with as little downtime as possible.

An hour of at-risk data certainly doesn't prevent us from "losing *any* data or losing *any* work." In addition, the restore scenario associated with this kind of backup plan is, as you shall see, hardly conducive to "as little downtime as possible."

Restore Scenarios

SQL Server recovery can be a time-consuming thing. Essentially, you have to start with your most recent full database backup, then add on the most recent differential and *every* transaction log backup made since then.

In fact, you have to be very specific about what you're doing when you conduct a restore—an aspect of SQL Server that I've frankly seen a lot of administrators mess up pretty badly. If you conduct a normal, full database restore, SQL Server will by default put the recovered database online as soon as it's done with the restore operation. If you still have a differential or some log backups to apply, you're out of luck; you have to *start the restore over.* The trick is to tell SQL Server, as you're restoring the full backup, that you have more files to restore. You continue telling it that until you restore the last transaction log backup, at which time you tell SQL Server that it's safe to start recovery. *Then* SQL Server will start applying the differential, then the transaction log backups, and *then* your database will be ready to use. "As little downtime as possible" isn't very little, in most cases, and you'll still be missing any changes that occurred after the most recent transaction log backup.

SQL Server Recovery

For a large database, SQL Server's recovery time can be quite lengthy. Let's say you use the backup plan shown in Figure 5.2, and something goes wrong at 4pm on Friday afternoon. You'll have a full backup from the prior weekend, Thursday night's differential—which may be quite large, since it contains *all* the changes from the full backup up to Thursday night—and hourly transaction log backups.

Not only do you have to wait for all those files to stream off tape or wherever you store them, you have to wait for SQL Server to work through them. It has to apply the differential backup to the full backup, then it has to replay *each individual transaction* from *every single transaction log*—in essence, it has to re-perform all the work that was done all day Friday. For a large, busy database, it may be a long time before the database is ready to use.

SQL Server doesn't natively support single-object restores. What you can do is restore a backup to a *different* database, then manually copy any objects you want restored from that backup. This lets you recover single stored procedures, tables, or even rows of data—provided you know how to do so manually.

SQL Server *does* support point-in-time recovery, with the obvious caveat that it can't restore to a point in time *later* than your most recent backup. Point-in-time recovery only works with transaction log backups because transactions in the log are time-stamped. If you discard Thursday's transaction log backups after making a differential backup on Thursday night, then the first point in time you can recover to is the time of that Thursday night differential. This actually makes backup management tricky because to enable maximum point-in-time recovery, you have to keep a *lot* of files hanging around: full backups, every night's differential, every hour's transaction log, and so forth.

Consider this scenario: You're using the example backup plan from Figure 5.2, which entails a weekly full, nightly differential, and hourly transaction log backups. Let's say you keep 3 weeks' worth of backups, and Week 3 is the most recent set. It's Friday afternoon, and you realize someone deleted a critical stored procedure. You need to recover the database to the *previous* Wednesday (Week 2) afternoon; Figure 5.3 illustrates the files that you have on-hand and which ones you'll have to restore.

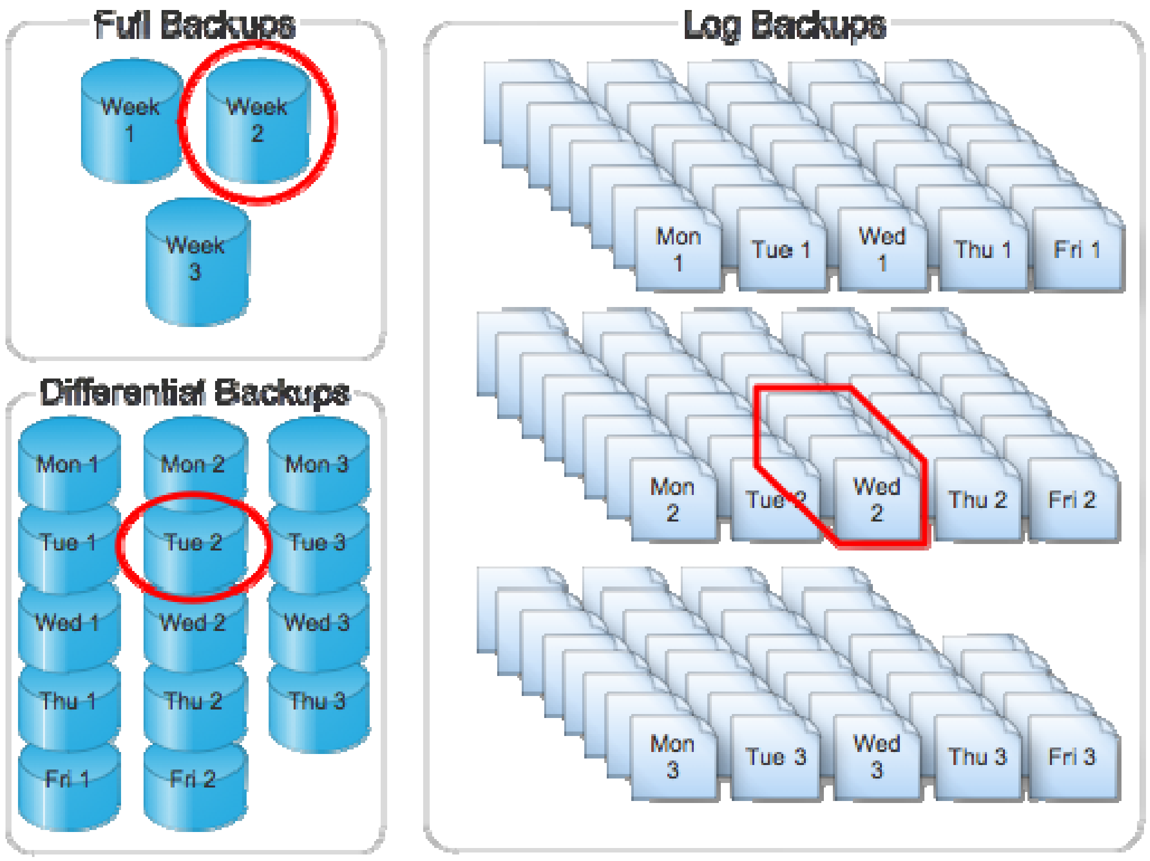


Figure 5.3: Sample recovery plan.

So, that's:

- The beginning-of-Week-2 full backup
- The Tuesday night differential from Week 2
- The Wednesday transaction log backups up to the point Wednesday afternoon you want to recover to

That's six or so files to recover, and then you wait for SQL Server to sort it all out. In total, you'll be keeping something like 140 files lying around, assuming you take a transaction log backup eight times a day (once every hour during the normal working day).

Disaster Recovery

SQL Server doesn't offer any kind of native disaster recovery capabilities. Essentially, if you lose an entire server, you'll have to recover Windows, install SQL Server, and then start restoring SQL Server backups to bring your databases as up to date as possible. Traditional third-party imaging software isn't effective because it's difficult to image an active SQL Server installation, and because imaging doesn't always work well with SQL Server's native backup capabilities—meaning it can be tricky to restore an image and then also restore normal SQL Server backups to bring your databases more up to date.

In short, let's hope you don't lose an entire SQL Server.

In fact, whole-server disaster recovery for SQL Server is so unsatisfying that Microsoft has made a considerable investment in SQL Server high-availability features that try to reduce the need to ever do a whole-server recovery. Some options include:

- **Transaction Log Shipping.** The idea here is to start off with two servers that have an identical copy of a database, then "ship" the transaction logs from the active server to the "hot spare" server. The spare re-plays the transactions to bring its copy of the database up to date; the theory is that if the main server dies, the hot spare can be brought it to replace it.
- **Database Mirroring.** Essentially the same idea as transaction log shipping, only the "hot spare" is kept more up to date and can take over automatically if the main server dies.
- **Clustering.** Utilizing Windows' native clustering capabilities, this provides a completely redundant server with direct access to the same database files as the "main" server.

All of these options require additional SQL Server installations and hardware (or virtual servers), and they're all designed to handle a complete-failure scenario; none of these actually provides for point-in-time recovery capabilities, so they're to be used *in addition* to normal backup techniques. It can get pretty expensive, especially for smaller and midsize companies who may not be able to afford this level of recoverability—at least in a Backup 1.0 world.

Backup Management

I touched on this earlier, but the short message is that SQL Server backup management can be pretty painful, unless you're only worried about restoring the database to its most recent state. In that case, you keep the most recent full backup, most recent differential, and all transaction logs since the differential; that's still a lot of files to maintain but it's a lot less than trying to keep a few weeks' worth of files.

I once had a job where we needed to be able to restore the database to any point in time *for 3 months*. You can imagine the number of files we had to maintain; I think it was close to 600 backup files, all floating around on different tapes, some of which had to be rotated off-site—it was a nightmare and just describing it is giving me unpleasant flashbacks. In fact, it was at that exact point in time that I started to realize that the Backup 1.0 way of doing things was *not* very efficient—especially because managing that many files *still* left us at-risk for an hour or more of data and work.

Rethinking Server Backups: A Wish List

So how can Backup 1.0 be improved from a SQL Server perspective? There's certainly plenty of room for improvement based on the traditional techniques and approaches I just discussed.

New and Better Techniques

The whole idea of being able to make transaction log backups to have less data at-risk is wonderful, but it is ultimately a *kludge.* It's a workaround to the snapshot-oriented approach of Backup 1.0; I've said it before and I'll say it again here: *Backups should be continuous.* It's not practical to continually make transaction log backups, and that's the best SQL Server can offer; that means we have to move *outside of the native APIs.* That's scary, I know, because so much of SQL Server depends on folks using those native APIs. But stick with me.

If we acknowledge that SQL Server's native APIs aren't going to give us frequent-enough backups, we need to look at other ways of getting to the data. Going through SQL Server *is not the answer* because SQL Server doesn't have the bandwidth to feed us any kind of continuous data stream. Instead, we need to grab that data directly from the *operating system (OS)*, as the data hits the disk. Keep in mind: As complicated as SQL Server is, ultimately it's all just bits on disk. There's no reason we couldn't have a Backup 2.0-style agent sitting on the SQL Server computer, grabbing disk blocks as SQL Server writes changes to the disk.

Clever readers will have spotted a problem with this theory, from my explanation on how SQL Server works:

> When SQL Server is told to do something, it's by means of a query, written in the Structured Query Language (SQL) syntax. In the case of a "modification" query, SQL Server modifies the pages of data in memory. **But it doesn't write those modifications back out to disk yet**, as there might be additional changes coming along for those pages and the system load might not offer a good disk-writing opportunity right then.

Oops. If SQL Server doesn't write the data to disk quickly, then that data is at-risk because all we're grabbing are the changes that actually make it onto the disk. But the answer to this potential problem also lies in the very way that SQL Server works:

> What SQL Server does do, however, is make a copy of the modification query in a special log file called the transaction log. This file, which has an .LDF filename extension, keeps a record of every transaction SQL Server has executed.

The transaction log itself is just a file on disk; Microsoft knows perfectly well that any data living entirely in memory is always at-risk, and so the transaction log's entries are written to disk immediately. All our agent would need to do is *also* grab the changes to the transaction log. Then, in a failure, we'd simply restore the database, restore the transaction log, and let SQL Server's nature take its course.

Better Restore Scenarios

The restore scenarios in a Backup 2.0 world would be vastly improved. For one, the concept of Backup 2.0 involves continuously streaming changed disk blocks to some central repository; that being the case, you'd simply select the *exact point in time* you wanted to restore to, then stream those disk blocks right back to where they came from. You might have to shut down SQL Server while you did that, but you might not; programmers can get pretty clever at manipulating SQL Server, and SQL Server itself is pretty open to manipulation in this regard.

Suddenly, no more worrying about full backups, differentials, and transaction logs. You don't care about the files per se; you only care about the disk blocks from the active database and log files. You're not making backups in the SQL Server sense of the term; you're actually just putting the computer's disk back to the condition it was at a certain point in time. SQL Server never actually enters its "recovery mode," because you've not restored any files in the SQL Server fashion. SQL Server simply resumes working with the database and log files just as it normally works with them.

This entire idea, which is at the heart of Backup 2.0, took me a while to really sort out in my mind. In the end, *everything we know about backups is wrong,* which is why I chose to use the term "Backup 2.0." This is an entirely different way of looking at things.

What about single-object recovery? That would still be tricky. Backup 2.0 will certainly let us restore a single database, rather than an entire server, if desired. But keeping track of which disk blocks within a database file go with a particular stored procedure, for example—that would probably be impossible. It certainly sounds difficult. But a Backup 2.0 solution *should* allow us to quickly restore a database to a different location—after all, a database is just a bunch of disk blocks, and they shouldn't care where they wind up—and we could use SQL Server's native tools to script out a stored procedure and then run that script on our production database, or use SQL Server tools to just copy database objects like users or whatever from one database to another.

Single SQL Objects: Tricky, Tricky

Part of the reason it's so tricky to recover a single SQL Server database object is that SQL Server stores objects—like stored procedures—as text definitions within a set of system tables inside the database. In other words, objects are externally indistinguishable from data; in most regards, objects *are* data.

A useful tool to have handy, then, is some kind of SQL Server comparison tool; use your favorite search engine to look for "sql server diff" and you should find several. These tools compare two databases—like a restored database and an in-production version—and show you the differences. In most cases, differences can be "forwarded" into the other database, making it easier to spot the exact differences between a restored database and its live counterpart, and to "restore" specific objects from the restored database into the live database.

Some Backup 2.0 toolsets might even include such comparison utilities, and might even incorporate them into the recovery process for a more seamless experience when you're just looking for a single object out of a backup.

Better Disaster Recovery

There's no doubt that Backup 2.0 can offer a better disaster recovery option than more traditional techniques. Just consider Figure 5.4, which compares the two philosophies in a practical disaster recovery timeline.

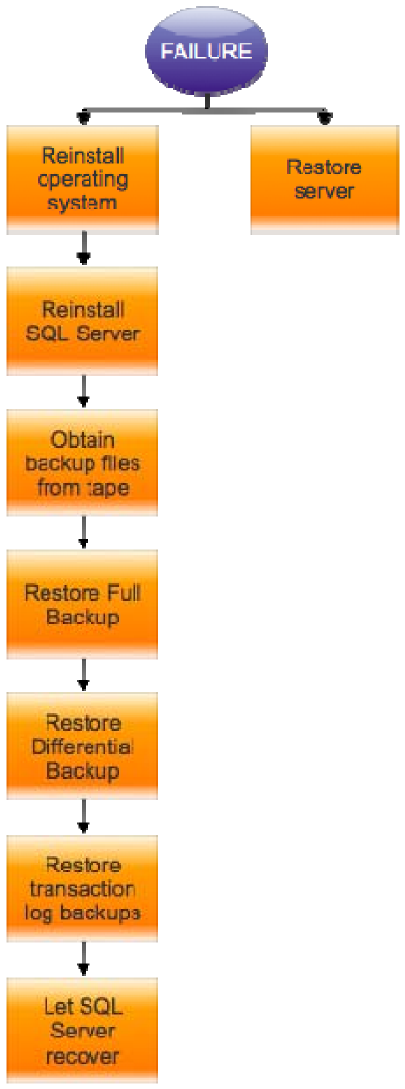

FAILURE
Reinstall operating system
Restore server
Reinstall SQL Server
Obtain backup files from tape
Restore Full Backup
Restore Differential Backup
Restore transaction log backups
Let SQL Server recover

Figure 5.4: Comparing disaster recovery techniques.

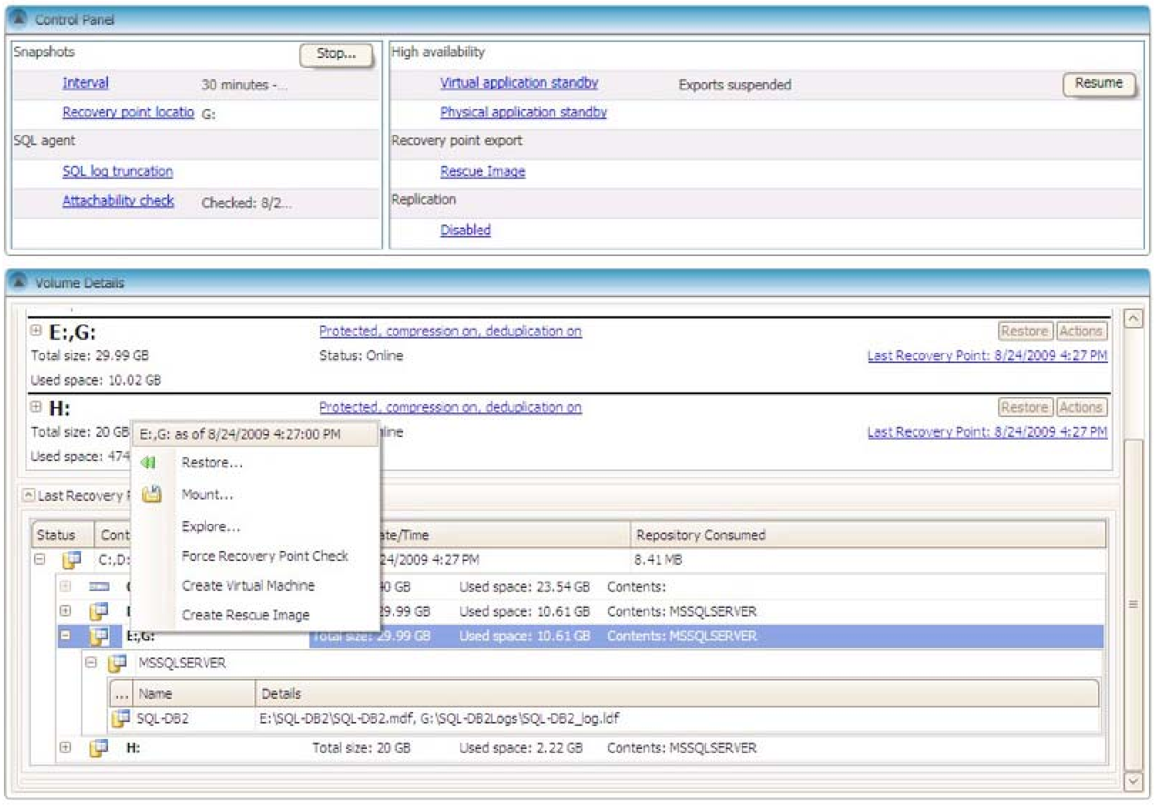
Backup 1.0 is on the left, where you're spending a ton of time manually recovering software and letting SQL Server deal with its backup files. Backup 2.0 is on the right, where you're simply pushing *all* the server's disk blocks back to the server's disk—recovering the OS, SQL Server, data files, and everything all at once, and to a specific point in time.

Now, I do realize that many third-party backup solutions of the Backup 1.0 variety *do* make backups of the *entire* server, and that many of them offer bootable CDs or DVDs that can kick-start a whole-server recovery. That's great—except that it still leaves you "recovered" to an old snapshot. Making a backup of an entire server is even more time consuming than backing up a single large database; you're less likely to have an up-to-the-minute backup, meaning more data is at risk and more time will be spent during a restore.

Another advantage of the Backup 2.0 technique—as I've pointed out in previous chapters—is that disk blocks *really don't care where they live.* Disk blocks could be restored to a different server, if the original one's hardware was irrecoverable. Disk blocks could be written to a *virtual* server, giving you a fantastic option for off-site recovery in the event of a data center disaster, like a flood or loss of utility power.

Easier Management

Management, for me, is where Backup 2.0 really has a chance to shine. Having managed the 600-ish files involved in a previous company's SQL Server backup plan, I love the way that Backup 2.0 doesn't focus on specific point-in-time snapshots. That means no managing backup files. Instead, you manage a single backup *repository*, where all the backed-up disk blocks live. Rather than juggling files and tapes, you use a centralized management console—like the one shown in Figure 5.5, for example—to manage the entire repository, which might well handle backups for many, many servers. You select the server you need to restore, select the disk volume that contains the files you want to restore, and indicate the point in time you want to restore to. The repository figures out which disk blocks are involved in that recovery operation, and streams them to the server you designate. Anything from a single file to an entire server can be recovered in the same fashion.


Control Panel
Snapshots
Interval 30 minutes -...
Recovery point locatio G:
SQL agent
SQL log truncation
Attachability check Checked: 8/2...
High availability
Virtual application standby Exports suspended
Physical application standby
Recovery point export
Rescue Image
Replication
Disabled

Volume Details
E:,G: Protected, compression on, deduplication on
Total size: 29.99 GB Status: Online Last Recovery Point: 8/24/2009 4:27 PM
Used space: 10.02 GB
H: Protected, compression on, deduplication on
Total size: 20 GB Last Recovery Point: 8/24/2009 4:27 PM
E:,G: as of 8/24/2009 4:27:00 PM

Status	Cont	ate/Time	Repository Consumed
	C:,D:	24/2009 4:27 PM	8.41 MB

	0 GB	Used space: 23.54 GB	Contents:
	9.99 GB	Used space: 10.61 GB	Contents: MSSQLSERVER
E:,G:	Total size: 29.99 GB	Used space: 10.61 GB	Contents: MSSQLSERVER
H:	Total size: 20 GB	Used space: 2.22 GB	Contents: MSSQLSERVER

MSSQLSERVER

Name	Details
SQL-DB2	E:\SQL-DB2\SQL-DB2.mdf, G:\SQL-DB2Logs\SQL-DB2_log.ldf

Figure 5.5: Examining the Backup 2.0 repository.

Easier management—letting the software juggle the backup data—is one of the real advantages that can be realized when we start rethinking what backups are all about.

SQL Server-Specific Concerns

So how will Backup 2.0 help address some of the concerns that are unique to SQL Server? Obviously, it depends on the exact Backup 2.0-style solution you're talking about, but there are certainly ways in which solution vendors could handle SQL Server issues.

Sprawl

Sprawl isn't a problem with Exchange Server or SharePoint; those applications live in the data center. SQL Server, however, spreads throughout the organization in desktop-level installations where users might not even realize that their data is contained in SQL Server. Even with client-level backup agents, this SQL Server data often goes unprotected; client-level agents are usually designed for simple file-and-folder backup, and don't typically include a SQL Server-specific agent. The "hidden" SQL Server instances run continuously, just like any installation of SQL Server, thwarting simple file-and-folder backup schemes at the client level.

Backup 2.0 can help. Because the whole idea of Backup 2.0 is based on capturing disk blocks, it can wedge itself into the file system at a very low level, using built-in Windows hooks designed for exactly this sort of activity. Figure 5.6 illustrates where a Backup 2.0 agent can work its way into the system while only adding 1 to 2% of overhead to the client system. This low level of overhead makes this type of agent perfectly suitable for workstations running "Express" or desktop instances of SQL Server.

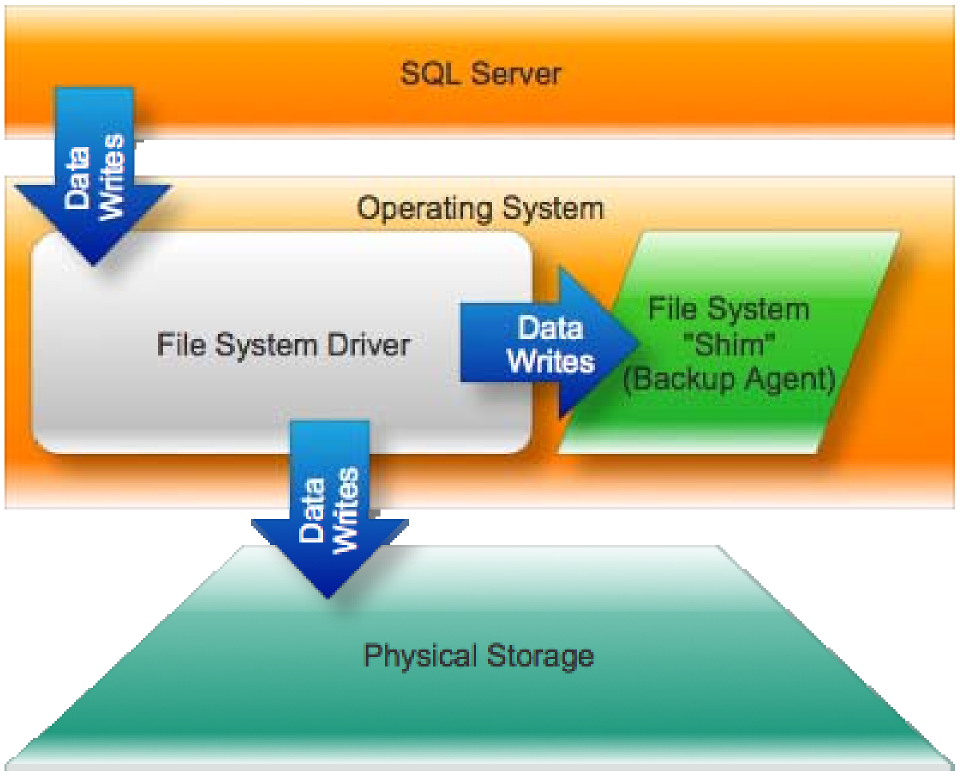


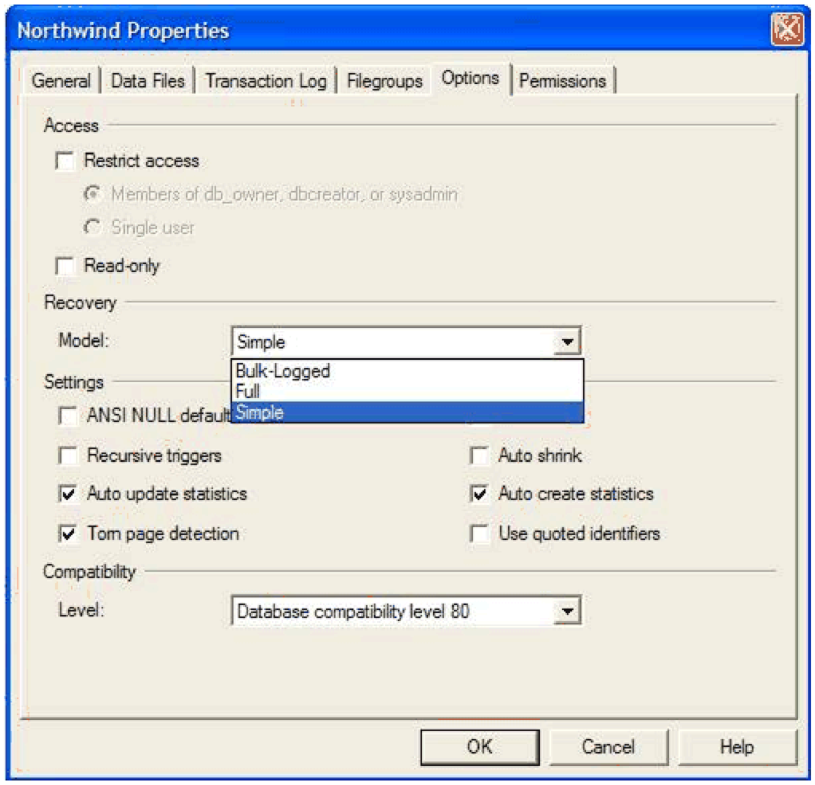
Figure 5.6: How backup 2.0 fits in.

Here's how I envision it working: A Backup 2.0 agent, written as a file system "shim," registers itself with the OS. When SQL Server saves data to disk—whether to a database file or to a transaction log—the shim is notified by the file system. As the file system writes the data to physical storage, the shim can read the newly-written blocks, compress them, and send them across the network to a central repository.

It's far more efficient, especially for client computers, than snapshot backups. Workstations running an "Express" edition of SQL Server typically have fairly low SQL Server workloads, since SQL Server is really serving only a single application in use by one or a few users. Rather than laboriously backing up the entire system or every database file every so often, Backup 2.0 just streams the few disk blocks that have changed. In a "sprawl" environment, it's the perfect way to create consolidated SQL Server backups—for all that important data that's living in all your "stealth" SQL Server installations.

Log Truncation

If Backup 2.0 is just capturing disk blocks, when does the SQL Server transaction log get truncated? Well, in some instances you might think you could just stop using the transaction log. As Figure 5.7 shows, a SQL Server database *can* be configured to use a "Simple" recovery model. Unlike the "Full" model, which operates as I described earlier in this chapter, the "Simple" model basically truncates the log as soon as a transaction's changes have been written to disk. In other words, the transaction log still exists, but it's not a recovery option because it won't ever contain very many transactions—it'll only contain those transactions whose data pages are still in memory.



Restrict access
Members of db_owner, dbcreator, or sysadmin
Single user
Read-only
Model: Simple
ANSI NULL default
Recursive triggers
Auto update statistics ✓
Torn page detection ✓
Auto shrink
Auto create statistics ✓
Use quoted identifiers
Level: Database compatibility level 80

Figure 5.7: Configuring a database for simple recovery model.

At first glance, it might seem like this would be perfect in a Backup 2.0 world. After all, you still get transaction log recovery for an unexpected server power outage, but the transaction log is self-maintaining and doesn't need to be truncated. Your Backup 2.0 solution is grabbing disk blocks almost in real time, and shipping them off to a backup repository—so what good is the transaction log?

In some scenarios, I'd say "go with Simple recovery!" But in others, I still like to have the piece of mind that the transaction log offers—and so I'd look for a Backup 2.0 solution that had the *ability to truncate the log* just as SQL Server does when it makes a successful backup. In other words, if the backup solution isn't using native SQL Server APIs—which *do* truncate the log after a successful backup—then the backup solution should fully replace those APIs, including log truncation capability.

Single-Object Recovery, Corruption, and Off-Location Restores

Single-object recovery, corruption, and off-location restores might seem like three pretty random topics to throw together, but they're all potential issues that are solved by the same thing. As I've described earlier in this chapter, single-object recovery in SQL Server pretty much always involves restoring the database to a different location, then copying objects from the restored database to the production database. That's just inherent in the way SQL Server works. The problem is that restoring a backup, as we've discussed, can take a lot of time. Spending hours restoring files just to grab a single accidentally-deleted stored procedure or view is *painful* and unrewarding in the extreme.

There are many other reasons to restore a database to a different location, which is also called an *off-location restore* or *alternate-location restore.* One reason is to compare a backed-up database to the current, live database, using some kind of comparison tool. Another reason might be to access data that was deliberately purged from the live database. Typically, the database is restored, from backup, to a different server, or to the same server under a different database name—both of which require "scratch space," or temporary space to hold the entire restored database for however long it is needed. Then, of course, there's also the time to restore all the database files and let SQL Server process them.

My third issue is one of backup data corruption, which is an insidious thing we've all run into: "The backup tape is corrupted!" Although Backup 2.0 relies less (or not at all) on tapes, data corruption is still a real concern, and any decent backup solution will offer ways to detect corruption.

All three of these issues—off-location restores, single-object recovery, and data corruption—can be handled by a single feature we can add to our Backup 2.0 spec: I propose calling it *backup mounting.* Think of it like this: Our backup repository contains a bunch of disk blocks, each one time-stamped to let us know when it was captured. There's really no reason we couldn't select a bunch of disk blocks from a given point in time, feed them to a specialized file system driver, and "mount" them like a normal disk volume. Figure 5.8 shows what I mean.

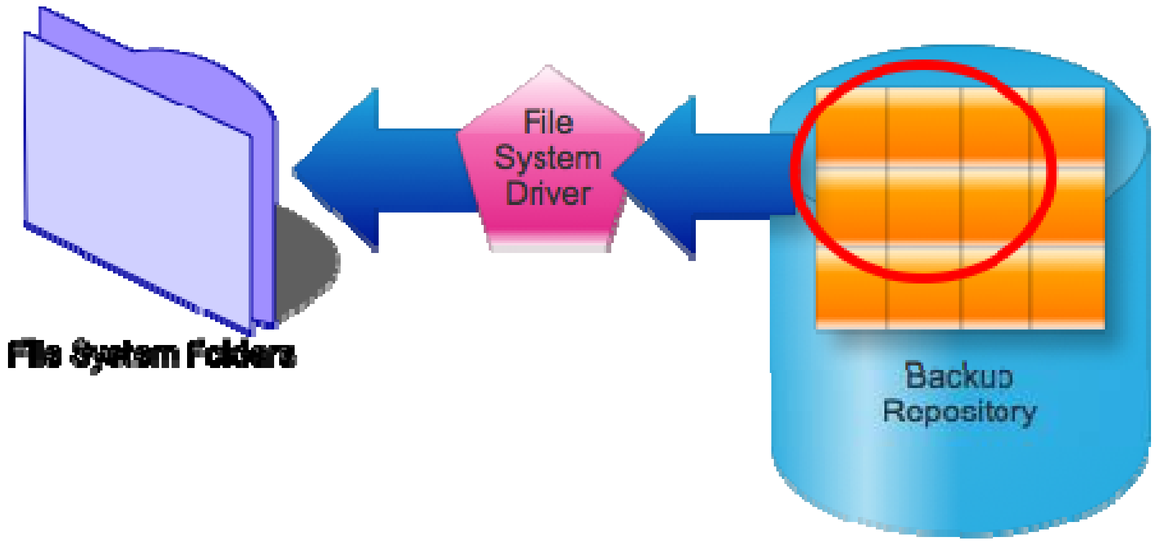


Figure 5.8: Mounting backed-up disk blocks as a disk volume.

A file system drive's job is to take some form of storage and make it look like a disk volume, files, and folders. Microsoft basically does this same thing in Windows 7, where the OS allows you to mount a virtual machine image as a disk volume. I'm simply proposing that Backup 2.0 include a special file system driver that lets Windows "see" selected portions of the backup repository as if they were a real, live, read-only disk drive.

Once that's accomplished, the backup solution can make SQL Server database and log files available *without performing a restore.* So in *zero time,* your database files could "appear"—in read-only form, of course—and be attached to a live instance of SQL Server. This would enable the backup solution to conduct "attachability" tests, to determine whether the backed-up image could be operated as a full database. If it couldn't, then corruption would be suspect. You could also attach backed-up databases on-demand for comparison purposes or for single-object recovery—*without ever having to restore anything.* You stay more productive, you don't need "scratch space," and you can "attach" a version of the database *from any point in time.* Truly a remarkable set of capabilities from a fairly simplistic notion—which is really what Backup 2.0 is all about: Simple, new notions that radically change the way we work, for the better.

Coming Up Next…

The last major server product I have to cover is SharePoint, which offers its own unique challenges. In fact, SharePoint—which has rapidly grown in popularity in the past few years—may be the biggest challenge that Backup 2.0 has to face. As I have in this and the previous chapters, I'll look at native solutions, cover problems and challenges, and compare the "1.0" way of doing things with a more enlightened "2.0" approach.