TECHNICAL BRIEF

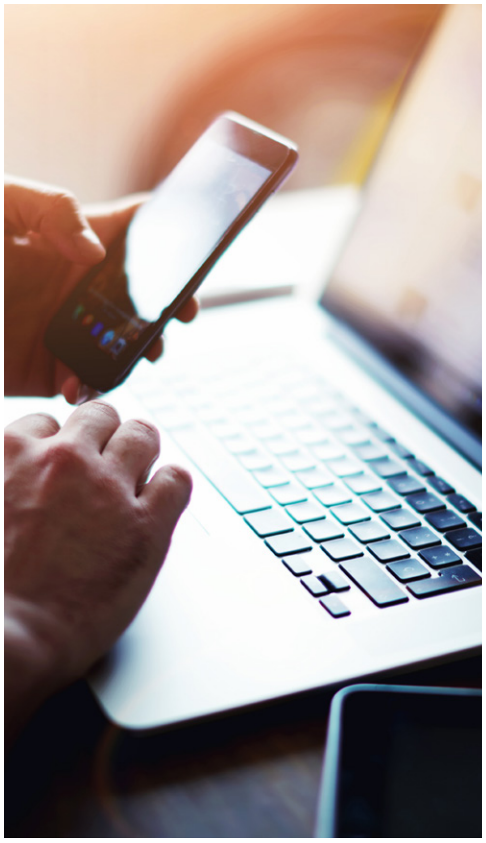
Data Integrity During a File Copy

Datadobi

Table of Contents

Data Integrity During a File Copy

INTRODUCTION



One of the critical activities that must be performed during a data copy is to validate the accuracy of the copied data. After all, what is the use of copied data if its integrity can't be proven?

It's a common misconception that using legacy copy tools such as Robocopy and rsync automatically validate the integrity of the data. As you will discover in this technical brief, this is both an erroneous and dangerous assumption. Because of the substantial manual effort and risk involved, rarely are migrations or replications that use legacy tools properly validated.

Through Datadobi's extensive experience in data copy, it's clear that during the course of its journey, there are many opportunities for corruption to occur on migrated or replicated data.

One of the key tenets of the DobiMiner® Suite is that every single file copied, whether for migration or replication, must be validated to prove that no corruption has occurred during the transfer. This is done with the use of hashes that are built into the workflow and run automatically.

In addition to explaining why legacy tools do not properly validate data and how DobiMiner does it differently, this technical brief will provide an in-depth explanation of hashing, and explain why network protocol integrity checking is a false hope when ensuring the integrity of copied data.

WHY LEGACY TOOLS DO NOT VALIDATE DATA

The two most common copy tools, Robocopy and rsync, are discussed here but the subject also applies to most other common copy utilities.

ROBOCOPY

Robocopy does not natively check the integrity of files written to a destination system. Rather, the utility Microsoft® FCIV (File Checksum Integrity Verifier) utility must be used. The FCIV utility must be downloaded from Microsoft and is an unsupported utility. It can generate both MD5 and/or SHA-1 hashes of filesystem object names provided as input.

Use of FCIV beyond testing a single file requires scripting. This means creating one set of scripts to execute the Robocopy sessions, another set to mine the logs for progress and/or errors, and another to run the FCIV validation checks. In addition, since the validation scripts would have to be run separately on both the source and destination systems, even more scripting would be required to take the output from the FCIV operations and compare the results. If failures are found, either manual effort to recopy the failed filesystem objects would be required, or further scripting would be needed to take the list of failures and use them as additional input into the Robocopy sessions.

RSYNC

Originally introduced in 1996, rsync brought a creative approach to copying filesystem objects between Linux/Unix systems. But the approach taken by rsync has led to a critical misunderstanding about its use of hashing.

When filesystem objects are being copied between systems, rsync will break a file into “chunks” (basically a series of blocks) and will calculate a simple 32-bit rolling checksum as well as a stronger 128-bit MD5 hash on the same block. These values are used to determine whether the same chunk of data already exists on the destination system and, if not, that it can be copied to the destination.

The important point about this process is that rsync is using checksums and hashing to determine what to copy, but it is not using the checksums and hashes to validate the integrity of the data. After data is copied to a destination system, entire filesystem objects are not read back – nor is an additional hash digest calculated from either the source or destination. This means that no comparison of the copied file is ever made. In effect, when rsync writes data, it is assuming that the receiving device commits the write with no errors. The rsync utility will not re-read the data and compare against the source.

If validation of the previously written filesystem objects is required, then either the *nix utilities md5sum or sha[1/224/256/384/256]sum must be used. Use of md5sum or sha[n]sum beyond testing a single file is a scripting exercise. This means creating one set of scripts to execute the rsync sessions, another set to mine the logs for progress and/or errors, and another to run the MD5 or SHA validation checks. Since the validation scripts would have to be run separately on both the source and destination systems, even more scripting would be required to take the output from the MD5 or SHA operations and compare the results. If failures are found, either a manual effort to recopy the failed filesystem objects would be required, or additional scripting is needed to take the list of failures and use them as additional input into the rsync sessions.

For both Robocopy and rsync, because any validity checking can only be done once all the data is copied, validation can only be carried out during a cutover window. This has the effect of reducing the amount of data that can be cutover during the allotted time, and even potentially derailing a costly cutover event completely if errors cannot be resolved in time.

This risk, combined with the substantial manual scripting work required, means that it is rare that any project of more than a few terabytes is ever validated properly – leaving a company open to the possibility of data loss during a migration or replication.

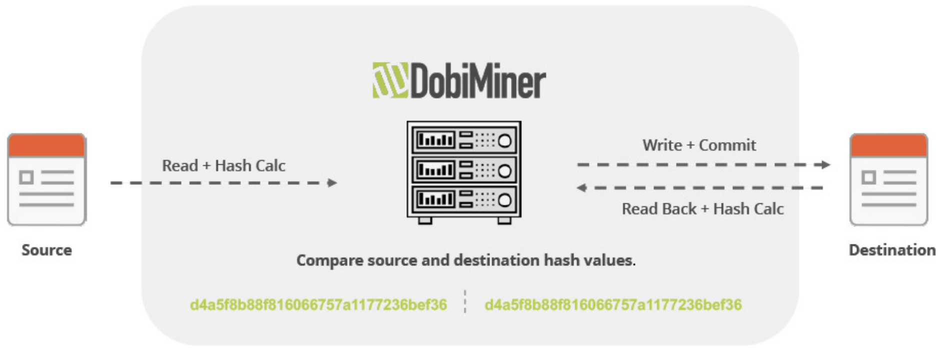
This risk, combined with the substantial manual scripting work required, means that it is rare that any project of more than a few terabytes is ever validated properly – leaving a company open to the possibility of data loss during a migration or replication.

WHAT DOBIMINER DOES DIFFERENTLY

In addition to many other differences with legacy tools, DobiMiner takes a completely different approach to data integrity that is both built into the workflow and is not optional. DobiMiner automates all integrity checking activities, thereby eliminating the scripting cycle of create, debug, modify, test, and repeat.

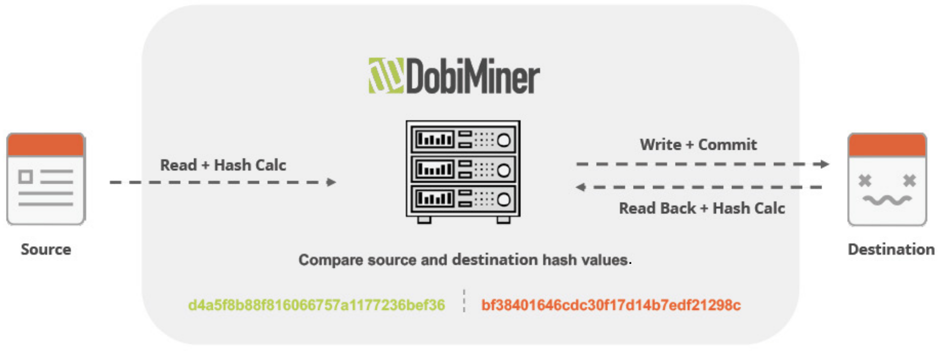
When DobiMiner reads a filesystem object from the source system, an MD5 hash is calculated. After the file is copied and committed to the destination system, the file is re-read and a second MD5 hash is calculated. The two hashes must then match in order to verify that the file has been copied and written to the destination system successfully.

FIGURE 1A Data copy validated by matching MD5 hash values


DobiMiner
Read + Hash Calc
Write + Commit
Read Back + Hash Calc
Source
Destination
Compare source and destination hash values.
d4a5f8b88f816066757a1177236bef36
d4a5f8b88f816066757a1177236bef36

Hashes match, data is valid.

FIGURE 1B Data copy invalidated by non-matching MD5 hash values


DobiMiner
Read + Hash Calc
Write + Commit
Read Back + Hash Calc
Source
Destination
Compare source and destination hash values.
d4a5f8b88f816066757a1177236bef36
bf38401646cdc30f17d14b7edf21298c

Hashes do NOT match, data is NOT valid.

In FIgure 1B, the difference shows an inconsistency between the two filesystem objects, meaning that a re-copy is required. Either some portion of the data was changed or corrupted in transit through the network, or there was potentially a bad disk write on the destination system.

Any difference, no matter how small, between the two filesystem objects will cause a mismatch between the hash values – and this will qualify as a copy error. DobiMiner will automatically re-copy the file during the next copy iteration until it is able to copy and validate the file successfully.

All errors are logged at the file level so that administrators are aware of the issue and can take action if needed. Upon final copy (cutover), a report can then be generated providing the hash value for every single file copied.

The rest of this technical brief provides in-depth information on the use of hashing, along with a section on the limits of network protocol integrity checking.

WHAT IS A HASH?

Calculating a hash is essentially the process of computing a fixed summary value based on an arbitrarily long input value. From a cryptographic viewpoint, many security functions are implemented using hash calculations based on a mathematical algorithm using a constant input number that, if unknown, makes it very difficult to derive the original input number without knowing the data used to create the hash.

For example, take an input number of 8,443 and assume a simplistic hashing algorithm is limited to multiplying this input number by 387. In this case, the hash value is calculated as 8,443 x 387 generating the product of 3,267,441. It would be very difficult to determine which two numbers were used when generating the value of 3,267,441 – but if you were told that 387 was the "magic" number used in the hashing algorithm, you could easily reverse the hash and determine the original input number.

Real hash algorithms are much more complex but this is the basic principal at work.

TYPES OF HASHING ALGORITHMS

The exponential increase in compute power over time serves in some ways as a double-edged sword. While it is now possible to execute highly complex hashing algorithms very quickly, it is also the case that criminal elements have increased compute power available to them as they attempt to break the encryption provided by the hashing algorithm. To that end, many types of hashing algorithms have been created over time as compute platforms have evolved from 8-bit on up to current 64-bit architectures.

Some of the more common hashing algorithms available are MD (Message Digest) and SHA (Secure Hash Algorithm). Each one has different variants that require more or less compute power to execute the algorithm given variable input data.

MD5 (Message Digest 5) is an algorithm that results in a 128-bit (16 byte) string that will show as a 32-character-long hexadecimal number. Other MD algorithms are the older MD2 and MD4.

Note: Some less commonly-used hash algorithms are BLAKE, RIPEMC, Skein, SWIFFT, and others.

SHA (Secure Hash Algorithm) offers more complex hash digest generation and has several different outputs ranging from 160 bits up to 512 bits, with the larger outputs requiring more compute power and time to generate. The different outputs of SHA will be determined by the variant of SHA in use (i.e. SHA-1, SHA-256, etc.)

For encryption purposes, the newer, more complex algorithms are required, but they come at the cost of increased compute overhead, which means slower processing. While simpler algorithms may not be preferred for encryption purposes, they are still useful for validating two independent copies of data, and this is the use case we are focused on when copying data.

HASHING FOR ENCRYPTION VS VALIDATION

From a security standpoint, hashing has historically been used to ensure that communications between two endpoints are properly encrypted. In the context of data copy, it is important to verify and validate that the data being copied is identical and no corruption has occurred. The validation operations need to be not only fast, but accurate.

DobiMiner leverages the MD5 hashing algorithm to verify the accuracy of the copied data. MD5, while not valid from a cryptographic viewpoint, is one of the best options when validating file content between disparate systems. Computationally, it is very fast to generate hash values and the values generated are highly unlikely to suffer from collisions (see below). The beauty of a hash is that the value is always a 128-bit value (displayed as a 32-character hex value) despite the arbitrary size of the file input being evaluated. The hash value will always be unique for each encountered file.

HASH COLLISIONS

In any discussion involving hashing algorithms, the subject of collisions always comes up. Hash collisions can occur when two different inputs produce the same hash output when run through the algorithm. When using hashes to validate data located on two different systems, collisions are not a concern because we're taking a single discrete filesystem object from both the source and destination system as input for the hash calculations. Even if the same exact filesystem object (down to the bit level) is stored multiple times in different directory structures, the evaluation of those entities is on a 1:1 basis, and each will be validated independently.

According to a forensics review[1] performed by AccessData, the following statement applies to the use of MD5 collisions in a file comparison scenario:

The MD5 algorithm breaks a file into 512-bit input blocks. Each block is run through a series of functions to produce a unique 128-bit hash value for the file. Changing just one bit in any of the input blocks should have a cascading effect that completely alters the hash results. Furthermore, since the key size of 128 bits has 3.4×10^{38} possible combinations, the chance of randomly finding two files that produce the same hash value should be computationally impossible (Schneier, 1996).

[1]AccessData whitepaper, "MD5 Collisions: The Effect on Computer Forensics," April 2006. Available at https://ad-pdf.s3.amazonaws.com/papers/wp.MD5_Collisions.en_us.pdf

THE LIMITS OF NETWORK PROTOCOL INTEGRITY CHECKING

While network protocols have integrity-checking built into the core protocol functionality, strict reliance on these mechanisms still leaves the door open to undetected data corruption. Network protocols have no way to detect if a bad disk write occurred and invalidated some or all of a given filesystem object.

A detailed discussion of network protocol validation mechanisms is beyond the scope of this paper but, in general, ethernet provides transmission resilience by including a frame check sequence (FCS) in each frame being placed on the network. The FCS is a 32-bit cyclic redundancy check (CRC) based on the payload included in the ethernet frame. When using TCP/IP, the ethernet frame is embedded in both IP packets and TCP packets, both of which have header checksums included. The TCP/IP header checksums and ethernet FCS provide a relatively robust system of transmission validation. When the receiving side calculates checksums that do not match, then packets must be retransmitted.

While network protocols are generally highly resilient to data corruption, there are times when silent data corruption can occur during transmission between endpoints. There are issues an FCS can't detect, such as any 1-bit error or adjacent 1-bit errors. The TCP and IP header checksums are based on 16-bit ones complement sum of the data. Reordering of two-byte words cannot be detected (for example, 01 02 03 04 are reordered to 04 03 01 02). There are also other scenarios that can "trick" the checksum.

SUMMARY

Any data mobility operation involving network-based transfers of file data between storage platforms is susceptible to corruption. Legacy tools and network protocols are unable to provide a level of proof required by any credible enterprise of its data's integrity.

The DobiMiner Suite automatically, and as standard, uses hash digests to determine the validity of data that has been copied between storage platforms, and provides proof that no data corruption has occurred. If file corruptions are detected, the DobiMiner Suite will log the error condition and automatically re-copy the file(s) identified during validation testing. Only once the file(s) have been validated via the hash digest comparison can the data then be considered valid and ready for production use.

Contact Datadobi today to learn more about how we can help your company transition to new technology quicker and increase your ROI: info@datadobi.com

Would you like to know more?

Contact sales@datadobi.com.

Datadobi

Datadobi.com