Perforce Performance

Michael Shields
Performance Lab Manager
Perforce Software

March, 2007

Abstract

As the number of customers using Perforce increases, the collective wisdom continues to expand. While the vast majority of this wisdom recorded throughout the digital universe is accurate, there are a few misconceptions regarding Perforce performance. And there are at least a few tidbits that can affect Perforce performance that are not yet common knowledge.

This paper discusses some of the misconceptions and lesser known realities of Perforce performance. While discussing these, related Perforce Server internals are also discussed.

1 Introduction

There are several areas of Perforce Server performance in which misconceptions are not uncommon. And there are always additional facts that need to be absorbed. The topics of metadata locking, filesystem cache, exclusionary entries, and embedded wildcards will be discussed. The discussion will expose some of the misconceptions in these areas, and present a significant amount of factual information on the topics.

2 Metadata Locking

For every Perforce command, the Perforce Server uses portions of the metadata. The metadata is stored in the db.* files in the P4ROOT location (which can also be specified using the "-r" option on the p4d command line). As examples of the metadata contained in the db.* files, db.rev contains information about the revisions in the repository, db.change contains information about the changelists (both pending and submitted, db.changex also contains information about the pending changelists as a performance optimization), db.integed contains information about the integration history, and db.have contains information about the revisions synced to each client workspace. The detailed schema of each db.* file can be found at:

`http://www.perforce.com/perforce/doc.current/schema/index.html`

Checkpoint and journal records also follow this schema.

Only the Perforce Server accesses the metadata directly. Perforce client applications, such as the Perforce Visual Client and the Perforce Command-Line Client, and client applications using a Perforce API, do not access the metadata directly. Rather, the client applications send commands to the Perforce Server which then accesses the metadata on behalf of the Perforce client

© 2007 Perforce Software

applications. After the Perforce Server has processed a command using the applicable metadata, the results are returned to the client application.

The Perforce Server can process multiple commands simultaneously. Concurrent processing of commands requires ensuring that each command uses metadata that is physically and logically consistent. That is, each command must use metadata in which the B-trees are correctly structured and the relationships between the different db.* files are as expected. For example, a reader must be assured that a B-tree it is traversing is not in the process of being restructured. A B-tree might be restructured by a writer splitting a page that was too full to accommodate inserted data. If a concurrent writer was allowed, the reader might miss metadata on pages being restructured. Further, a reader must be assured that, for example, the applicable entries in db.rev and db.revcx exist for submitted changelists in db.change. As a changelist is committed, the changelist in db.change is updated as submitted prior to writing the applicable db.rev and db.revcx entries. If a concurrent changelist commit was allowed, the reader might miss entries not yet written to db.rev and db.revcx.

Clearly, some method of synchronization between concurrent commands is required to ensure metadata consistency. One misconception is that there is a single global lock that controls all access to the metadata. One variant of this misconception is that the single global lock is an exclusive lock, allowing only a single command access to the metadata at a time. In reality, there is no single global lock.

Each command takes a lock on each db.* file that it needs to access. Depending upon the type of access to the db.* file that is needed, either a shared lock or an exclusive lock is taken on the db.* file. A shared lock on a db.* file allows concurrent access to the db.* file by other commands needing only a shared lock on the same db.* file. An exclusive lock on a db.* file prohibits concurrent access to the db.* file by other commands needing either a shared or exclusive lock on the same db.* file. Commands that might need to read from a db.* file but won't need to write to the db.* file take a shared lock on the db.* file. Commands that might need to write to a db.* file take an exclusive lock on the db.* file.

Operating system utilities such as "strace" on Linux platforms, "truss" on FreeBSD and Solaris, and "FileMon" on Windows can be used to observe locking behavior. The following "strace" example shows locks taken on the db.* files by the Perforce Server while processing a command. The Perforce Server is started using:

```
$ strace -etrace=open,flock,close p4d -f -p 50620
(system calls for server startup)
Perforce Server starting...
```

To simplify tracing the system calls, the Perforce Server is started single threaded using the "-f" flag.

The Perforce command is invoked from another session:

```
$ p4 -p 50620 sync //depot/main/... //depot/branch1/...
//depot/main/file1#1 - added as /p4client/depot/main/file1
```

© 2007 Perforce Software

```
//depot/main/file2#1 - added as /p4client/depot/main/file2
//depot/main/file3#1 - added as /p4client/depot/main/file3
//depot/branch1/file1#1 - added as /p4client/depot/branch1/file1
//depot/branch1/file2#1 - added as /p4client/depot/branch1/file2
//depot/branch1/file3#1 - added as /p4client/depot/branch1/file3
```

For this example, the "strace" utility reports the following (slightly annotated) "open()", "flock()", and "close()" system calls:

(begin command startup phase)

```
open("journal", O_WRONLY|O_APPEND|O_CREAT|O_LARGEFILE, 0666) = 5
open("db.depot", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 6
flock(6, LOCK_SH)                       = 0
flock(6, LOCK_UN)                       = 0
open("license", O_RDONLY|O_LARGEFILE)   = 7
close(7)                                = 0
open("db.group", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 7
flock(7, LOCK_SH)                       = 0
flock(7, LOCK_UN)                       = 0
open("db.protect", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 8
flock(8, LOCK_SH)                       = 0
flock(8, LOCK_UN)                       = 0
open("db.user", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 9
flock(9, LOCK_EX)                       = 0
flock(5, LOCK_EX)                       = 0
flock(5, LOCK_UN)                       = 0
flock(9, LOCK_UN)                       = 0
open("db.domain", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 10
flock(10, LOCK_EX)                      = 0
flock(5, LOCK_EX)                       = 0
flock(5, LOCK_UN)                       = 0
flock(10, LOCK_UN)                      = 0
open("db.view", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 11
flock(11, LOCK_SH)                      = 0
flock(11, LOCK_UN)                      = 0
```

(end command startup phase)

(begin compute phase for first argument)

```
open("db.have", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 12
flock(12, LOCK_SH)                      = 0
open("db.resolve", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 13
flock(13, LOCK_SH)                      = 0
open("db.revdx", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 14
flock(14, LOCK_SH)                      = 0
open("db.revhx", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 15
flock(15, LOCK_SH)                      = 0
open("db.working", O_RDWR|O_CREAT|O_LARGEFILE, 0666) = 16
flock(16, LOCK_SH)                      = 0
flock(16, LOCK_UN)                      = 0
flock(15, LOCK_UN)                      = 0
flock(14, LOCK_UN)                      = 0
flock(12, LOCK_UN)                      = 0
```

© 2007 Perforce Software

```
  flock(13, LOCK_UN)                          = 0
```
(end compute phase for first argument)
(begin execution phase for first argument)
```
  open("depot/main/file1,v", O_RDONLY|O_LARGEFILE) = 17
  close(17)                                   = 0
  open("depot/main/file2,v", O_RDONLY|O_LARGEFILE) = 17
  close(17)                                   = 0
  open("depot/main/file3,v", O_RDONLY|O_LARGEFILE) = 17
  close(17)                                   = 0
```
(end execution phase for first argument)
(begin compute phase for second argument)
```
  flock(12, LOCK_SH)                          = 0
  flock(13, LOCK_SH)                          = 0
  flock(14, LOCK_SH)                          = 0
  flock(15, LOCK_SH)                          = 0
  flock(16, LOCK_SH)                          = 0
  flock(16, LOCK_UN)                          = 0
  flock(15, LOCK_UN)                          = 0
  flock(14, LOCK_UN)                          = 0
  flock(12, LOCK_UN)                          = 0
  flock(13, LOCK_UN)                          = 0
```
(end compute phase for second argument)
(begin execution phase for second argument)
```
  open("depot/main/file1,v", O_RDONLY|O_LARGEFILE) = 17
  close(17)                                   = 0
  open("depot/main/file2,v", O_RDONLY|O_LARGEFILE) = 17
  close(17)                                   = 0
  open("depot/main/file3,v", O_RDONLY|O_LARGEFILE) = 17
  close(17)                                   = 0
  flock(12, LOCK_EX)                          = 0
  flock(5, LOCK_EX)                           = 0
  flock(5, LOCK_UN)                           = 0
  flock(12, LOCK_UN)                          = 0
```
(end execution phase for second argument)
(begin command cleanup phase)
```
  close(4)                                    = 0
  close(5)                                    = 0
  close(9)                                    = 0
  close(7)                                    = 0
  close(6)                                    = 0
  close(10)                                   = 0
  close(11)                                   = 0
  close(12)                                   = 0
  close(13)                                   = 0
  close(14)                                   = 0
  close(15)                                   = 0
  close(16)                                   = 0
  close(8)                                    = 0
```
(end command cleanup phase)

© 2007 Perforce Software

This example shows that locks on the db.* files are taken (LOCK_SH or LOCK_EX) and released (LOCK_UN) as needed throughout the command, rather than holding the locks over the entire command. Perforce commands generally take and release locks as needed throughout the command. This example also shows that either a shared or exclusive lock is taken on a db.* file as needed by different portions of the command. And also apparent in this example, Perforce commands generally lock only a subset of the db.* files. One of the db.* files not locked in this example is db.rev, which is generally one of the more active db.* files. db.rev is not locked in this example because the arguments provided to the "sync" command specified only head revisions. When only the head revisions are needed, it is faster to scan them from db.revhx and db.revdx, which contain only the undeleted and deleted head revisions, respectively.

The example also shows the general structure of most commands processed by the Perforce Server. Each command has a startup phase, during which authentication occurs and a context within which the command will be processed is established. Most commands have both a compute phase and an execution phase for each argument specified in the command. The command's functionality is generally implemented in the compute and execution phases. And each command has a cleanup phase, during which any remaining resources are released and any remaining open db.* files are closed. Some commands, such as "submit", have additional phases.

The compute phase of most commands computes the dataset that will be passed on to the execution phase. Computing the dataset generally requires shared locks on the db.* files that might contain information relevant to the computation of the dataset. Depending on the command, options, and the argument being processed, the amount of data scanned in a db.* file might be significant. While scanning the data in a db.* file, not only is a shared lock held on the db.* file, but shared locks are also held on other db.* files that might contain information related to the computation. So an extended scan of data in a single db.* file can delay the release of shared locks held on several db.* files.

If a compute phase requires metadata from a remote server, shared locks might be held on some of the local server’s db.* files while the metadata is retrieved from the remote server. The duration of the shared locks is dependent upon the speed at which the metadata can be communicated over the network from the remote server to the local server. If there is an anomaly affecting the network between the servers, the shared locks might be held for an extended amount of time, which might affect concurrency on the local server. It is therefore recommended that, for best performance, the "remote depot" feature only be used for periodic drops of shared assets. Remote depots should be protected such that they can only be accessed by automation performing periodic drops.

Once the command's compute phase for the argument being processed has completed, the command's execution phase starts processing using the computed dataset. Depending on the command, options, and the argument, the Perforce Server might interact with a client application over the network during the execution phase. Generally, no locks are held on the db.* files while the Perforce Server does network I/O during the execution phase. This allows concurrent processing of other Perforce commands even if, for example, a problem occurs on a subnet that delays network I/O between the Perforce Server and a client application.

© 2007 Perforce Software

While interacting with the Perforce Server during the command's execution phase, a client application might acknowledge completion of an event that requires the Perforce Server update one or more of the db.* files. Examples include a client application acknowledging receipt of a revision synced for which the Perforce Server must update the client workspace's have list (the list of files synced to the client's workspace), or a client application acknowledging opening a file for integrate for which the Perforce Server must update db.working, db.locks, and db.resolve. When updating db.* files during the execution phase, the Perforce Server generally does no network I/O during the actual update. The db.* file updates are generally very quick, with the possible exception of during a very large changelist submission. The I/O subsystem speed can also affect the speed of the db.* file updates.

In the "strace" example earlier in this section, we saw several files synced to the client workspace during two execution phases. But there was only one update of the client workspace's have list, which occurred during the second execution phase. The single update of the client workspace's have list is the result of a Perforce Server optimization introduced in the 2003.2 release. The optimization buffers 100 acknowledgements from the client application of receipt of a revision synced before updating the client workspace's have list. And at the end of the sync's last execution phase, the optimization updates the client workspace's have list with any remaining acknowledgements buffered since it was last updated. A similar optimization was introduced in the 2005.2 Perforce Server that buffers 100 acknowledgements from the client application of opening a file for integrate (or branch) before updating db.working, db.locks, and db.resolve.

When taking locks on a number of db.* files during some portion of a command, the Perforce Server acquires locks in the order shown in the "Tables in Locking Order" section of the schema. Always acquiring locks in the same order eliminates the possibility of deadlock between two concurrent commands. Eliminating the possibility of deadlock results in less complexity in the Perforce Server. Less complexity can generally result in a faster and more stable product.

Always taking locks on a number of db.* files in the same order generally results in acceptable concurrency. But there is a subtle scenario in which concurrency can be less than desired. The scenario can occur when the Perforce Server needs to take exclusive locks on several of the db.* files for some portion of a command to continue. It might happen that needed exclusive locks on one or more of the db.* files are acquired, but another needed exclusive lock on a db.* file further along in the lock order cannot be acquired due to other commands holding shared locks on the same db.* file. The command holding exclusive locks on one or more of the db.* files stalls while waiting to acquire the needed exclusive lock. This command can be thought of as a "wedge", since it is wedged between commands with shared locks held on the db.* file for which it needs an exclusive lock, and commands that need locks on the db.* files for which it holds exclusive locks.

Because of the "wedge" scenario, a reader scanning a significant amount of data in a db.* file might indirectly block other readers. Readers with shared locks held on a db.* file for an extended amount of time while a wedged command is waiting for an exclusive lock on the same db.* file and holding exclusive locks on other db.* files will block other readers needing a shared lock on one of the other db.* files for which exclusive locks are held by the wedged command. This scenario is not a deadlock because the shared locks blocking the needed exclusive lock will

be released, and then the exclusive lock will be acquired and the command will then continue. A slow I/O subsystem might also contribute to encountering this scenario. Perforce Server development continues to implement changes that minimize the possibility and effect of the "wedge" scenario.

3 Filesystem Cache

It is certainly true that if active Perforce Server processes are swapped out of physical memory, then it is likely that the amount of physical memory is not adequate. But it is not true to infer that the amount of physical memory is adequate if active Perforce Server processes are not swapped out of physical memory. In addition to the memory required by active Perforce Server processes, Perforce Server performance is significantly enhanced by additional physical memory utilized by the operating system as filesystem cache. I/O requests that can be satisfied from the filesystem cache complete much faster than I/O requests that must physically transfer data from the disks. Faster I/O completion using the filesystem cache generally results in quicker processing of the various phases within the Perforce Server. Quicker processing usually decreases the amount of time that each phase holds locks on the db.* files needed for the phase. Locks on the db.* files that are not held as long generally leads to better concurrency, allowing the Perforce Server to process more commands faster.

Filesystem cache content generally consists of recently used pages from disk files. On the Perforce Server machine, a significant portion of the filesystem cache will be pages from the db.* files. The Perforce Server typically accesses metadata in a db.* file by traversing the B-tree structure within the file, rather than sequentially accessing all pages in the db.* file. Because the B-tree structure is typically used to access metadata in the db.* files, the filesystem cache typically contains only the recently used portions the db.* files. It is generally not true that one or more db.* files are wholly contained within the filesystem cache. Portions of the db.* files storing metadata for client workspaces that are not used, branches for releases retired long ago, and other infrequently accessed metadata will usually not be in the filesystem cache.

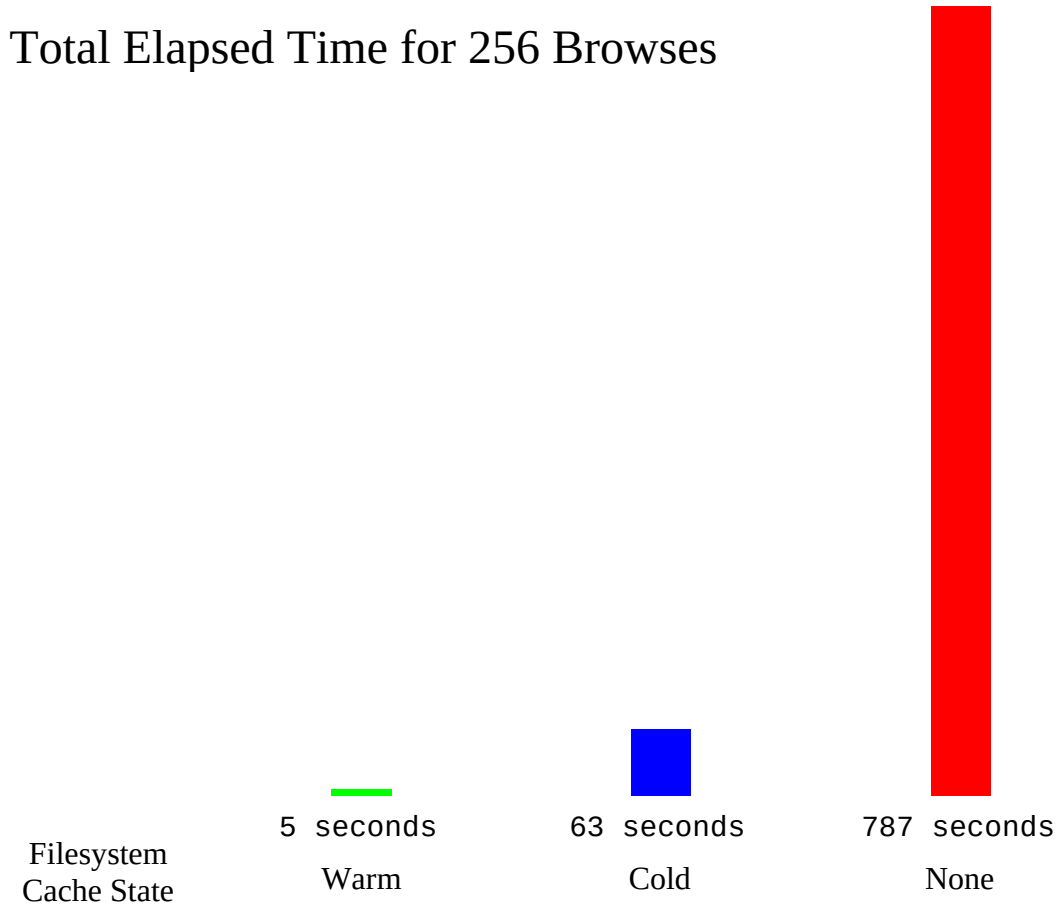
The Perforce Server relies heavily on the filesystem cache within the operating system since there is no shared cache implemented within the Perforce Server. Those familiar with database engine internals know that implementing a cache shared amongst multiple threads of execution can be difficult. Among the challenges of shared cache implementations is maintaining the integrity of the data while attempting to enhance performance. Because of the potential difficulties, the Perforce Server instead relies on the similar functionality of the filesystem cache. The filesystem cache implementation within most modern operating systems, together with private caching at several layers in each of the Perforce Server processes, is generally sufficient.

The performance benefit to the Perforce Server of an efficient filesystem cache can be significant. As an example, a benchmark was run using a Perforce Server on a Solaris 10 machine with three different filesystem cache states: none, cold, and warm. (The absence of a filesystem cache was benchmarked using the "forcedirectio" mount option on the Solaris 10 UFS filesystem where the db.* files were located.) For each run of the benchmark, four client applications executed concurrently from a Linux 2.6 machine over a dedicated subnet. Each client application randomly browsed the repository, using "dirs" and "fstat" commands typical of

© 2007 Perforce Software

those issued by the Perforce Visual Client as a user is drilling down through the repository structure. Each client application did 64 "browse" operations, without delay between browses. Each "browse" started at the top of the repository structure and drilled down until a file was selected. The client applications' pseudo-random number sequences were seeded using different numbers, and the same set of numbers were used to seed the four client applications in each of the three runs. So the three filesystem cache states were benchmarked using an identical random test.

The total amount of time for all four concurrent client applications finishing their 64 random browses (a total of 256 browses) was:



The above results show the benefit to the Perforce Server of an efficient filesystem cache, even if the cache is cold and the workload random. Machines with little physical memory in excess of that used by the active Perforce Server processes will have a small filesystem cache, resulting in performance nearly as abysmal as that shown above for no filesystem cache. It is therefore important to ensure that the Perforce Server machine is configured with an amount of physical memory that results in a filesystem cache large enough to minimize the number of I/O requests that must physically transfer data from the disks.

© 2007 Perforce Software

The amount of physical disk activity, particularly on a storage device containing only the db.* files, is a good indication of filesystem cache effectiveness. A low amount of physical disk activity while the Perforce Server is actively reading metadata indicates that the filesystem cache is large enough to accommodate the metadata actively used at the time. But a high amount of physical disk activity attributable to reading the db.* files might indicate that Perforce Server performance could benefit from more physical memory in the machine that the operating system would use as part of a larger filesystem cache. Operating system utilities such as "iostat" on Linux, FreeBSD, and Solaris platforms, and "perfmon" on Windows can be used to observe the amount of physical disk activity.

4 Exclusionary Entries

The "Tuning Perforce for Performance" chapter of the Perforce System Administrator's Guide discusses the performance benefits of limiting the number of files considered by a command. Two mechanisms discussed in the documentation for limiting the number of files are protections and client views. The type of protection and client view entries can have a significant impact on Perforce Server performance. The protection entries can have more of an impact since protection entries can affect multiple users, which typically implies multiple client workspaces.

Though somewhat counterintuitive, it is true that exclusionary protection and client view entries can cause the "dirs" command to scan more file entries in the metadata than might be expected. (Exclusionary entries have a "-" as the first character of the depot file path.) For each compute phase of a "dirs" command, the Perforce Server uses the non-exclusionary entries to position just prior to directories that will match the argument. As directories are collected, the Perforce Server efficiently "skips" from directory to directory. But if exclusionary entries are used, multiple file entries might be scanned following a position just prior to a directory. If the depot path in an exclusionary entry has a wide scope, the additional scanning can be significant, making the "dirs" command inefficient.

The following example demonstrates the efficient skipping of the "dirs" command. Given the dataset:

```
//depot/product/main/art/file1.jpg
                        file2.jpg
                        file3.jpg

//depot/product/main/doc/file1.fm
                        file2.fm
                        file3.fm

//depot/product/main/src/file1.cc
                        file2.cc
                        file3.cc
```

And given the protections table:

```
write group eng * //depot/product/main/...
```

For the compute phase of the "p4 dirs //depot/product/main/*" command, the Perforce Server initially positions just prior to the first directory that will match the "//depot/product/main/*" argument. The "//depot/product/main/art/file1.jpg" file entry is then scanned and the "//depot/product/main/art" directory is collected. The Perforce Server then positions just prior to the next directory that will match the argument. The "//depot/product/main/doc/file1.fm" file entry is then scanned and the "//depot/product/main/doc" directory is collected. The Perforce Server then positions just prior to the next directory that will match the argument. The "//depot/product/main/src/file1.cc" file entry is then scanned and the "//depot/product/main/src" directory is collected. The Perforce Server then positions just prior to the next directory that will match the argument. Attempting to scan finds no file entry matching the argument. The compute phase completes and the execution phase begins, during which the three directories collected during the compute phase are returned to the client application.

Continuing the example, suppose development has requested that their client workspaces not include documentation. One possible solution would be adding an exclusionary protection entry, resulting in the protections table:

```
write group eng * //depot/product/main/...
list group dev * -//depot/product/main/doc/...
```

This solution introduces some additional scanning when a developer issues the "p4 dirs //depot/product/main/*" command. After the "//depot/product/main/art" directory is collected, the Perforce Server then positions just prior to the next directory that will match the argument. The "//depot/product/main/doc/file1.fm", "//depot/product/main/doc/file2.fm", and "//depot/product/main/doc/file3.fm" file entries are then scanned and discarded due to the exclusionary protection entry. The scan continues through the "//depot/product/main/src/file1.cc" file entry and the "//depot/product/main/src" directory is collected.

Compounding the example, suppose a developer also desires that their client workspace not include the art assets. One possible solution would be adding an exclusionary entry in the client view, resulting in:

```
//depot/product/main/... //dev-client/depot/product/main/...
-//depot/product/main/art/... //dev-client/depot/product/main/art/...
```

This solution adds scanning as well. For the compute phase of the "p4 dirs -C //depot/product/main/*" command (note the addition of the "-C" flag, which limits the output using the client view), the initial position is followed by scanning the "//depot/product/main/art/file1.jpg", "//depot/product/main/art/file2.jpg", and "//depot/product/main/art/file3.jpg" file entries, which are discarded due to the exclusionary entry in the client view. With no directory yet collected, scanning continues. The "//depot/product/main/doc/file1.fm", "//depot/product/main/doc/file2.fm", and "//depot/product/main/doc/file3.fm" file entries are scanned and discarded due to the exclusionary protection entry. With no directory yet collected, scanning continues. The "//depot/product/main/src/file1.cc" file entry is scanned and not discarded, and the "//depot/product/main/src" directory is collected.

There are solutions that do not incur any additional scanning. Instead of adding the exclusionary protection entry so that documentation is not included in development's client workspaces, inclusionary protection entries for directories other than documentation are added for development. And the original protection entry for all of engineering is removed and added for engineering groups other than development. The resulting protection table is:

```
write group dev * //depot/product/main/art/...
write group dev * //depot/product/main/src/...
write group doc * //depot/product/main/...
write group qa * //depot/product/main/...
write group rel * //depot/product/main/...
```

Now when a developer issues the "p4 dirs //depot/product/main/*" command, the Perforce Server initially positions just prior to the first directory allowed by the applicable inclusionary protection entries matching the argument. The "//depot/product/main/art/file1.jpg" file entry is then scanned and the "//depot/product/main/art" directory is collected. The Perforce Server then positions just prior to the next directory allowed by the applicable inclusionary protection entries that match the argument. The "//depot/product/main/src/file1.cc" file entry is then scanned and the "//depot/product/main/src" directory is collected.

The developer that desired not including the art assets in their client workspace could also use inclusionary entries in the client view, rather than adding an exclusionary entry. The resulting client view is:

```
//depot/product/main/doc/... //dev-client/depot/product/main/doc/...
//depot/product/main/src/... //dev-client/depot/product/main/src/...
```

For the "p4 dirs -C //depot/product/main/*" command, the Perforce Server initially positions just prior to the first directory allowed by the applicable inclusionary protection entries matching the argument and also mapped by a non-exclusionary entry in the client view matching the argument. Though the "//depot/product/main/doc" directory is mapped by an inclusionary entry in the client view, there are no inclusionary protection entries that allow the developer access to the directory. So the "//depot/product/main/src/file1.cc" file entry is scanned and the "//depot/product/main/src" directory is collected.

This extended example shows that where possible, inclusionary entries should be used instead of exclusionary entries. As the scope of the depot path in an exclusionary entry widens, the additional scanning in the "dirs" command can make the command inefficient. When the scope of the depot path in an exclusionary entry is sufficiently narrow, the performance impact can be minimal. But since the "dirs" command is routinely issued by the Perforce graphical client applications, the use of exclusionary entries should be minimized.

5 Embedded Wildcards

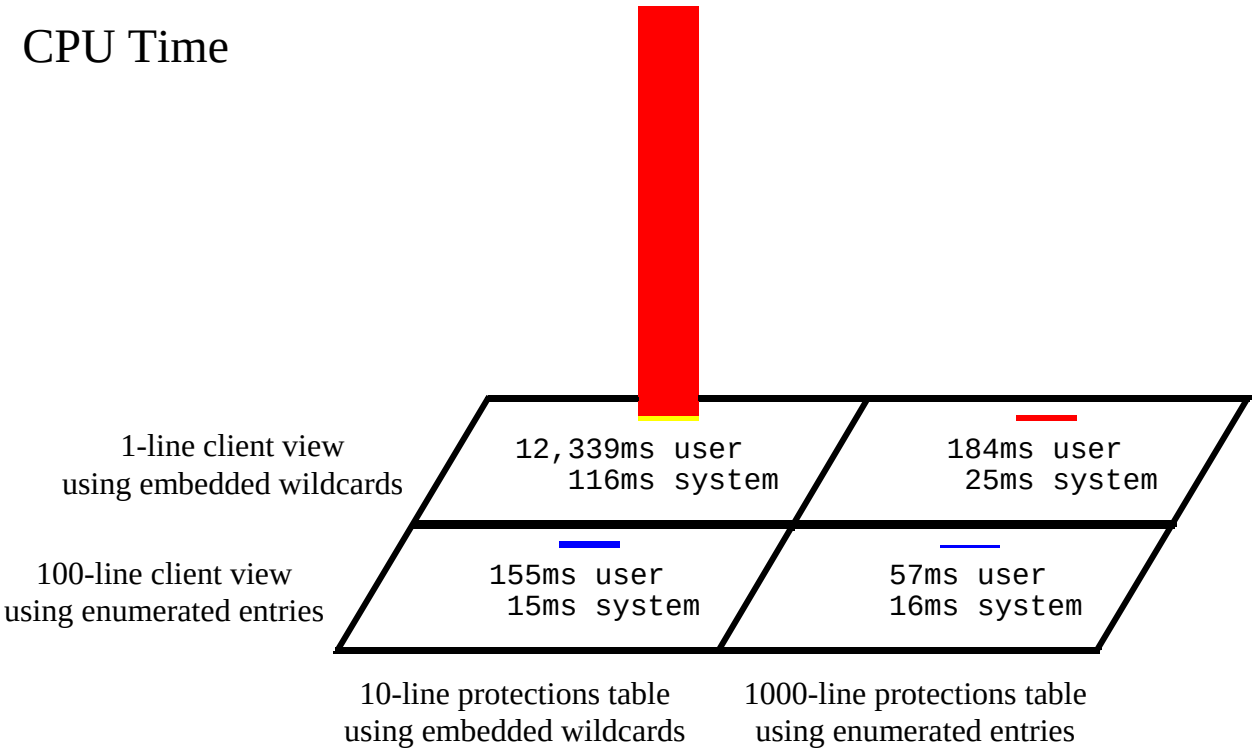
Paths with embedded wildcards, such as "//depot/main/.../doc/...", can be used to construct a more concise protections table and views. But while a concise protections table and views might

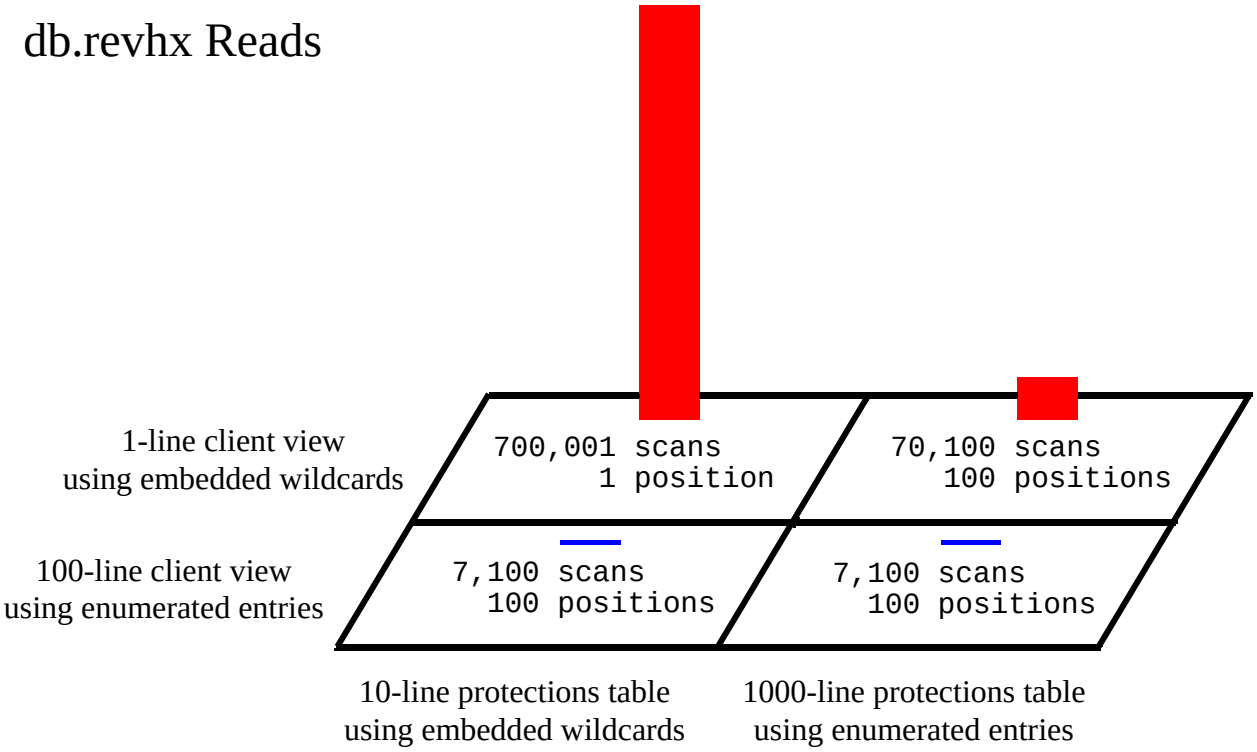
be easier to understand, it is also true that they might not result in the best Perforce Server performance. When paths with embedded wildcards are used in both the protections table and a view that are joined together, performance can degrade. For best performance, the use of paths with embedded wildcards should be minimized in both the protections table and views.

There are at least two solutions to widespread use of paths with embedded wildcards. The need for paths with embedded wildcards might be the result of a repository structure that is less than optimal with respect to its use. Examples might be documentation directories that are scattered within a source code tree, or platform-specific directories that are scattered within a generic tree. The more difficult solution (though perhaps with the greater benefit) might be restructuring the repository after a thorough analysis of its intended use.

Another possible solution is replacing the entries that use paths with embedded wildcards. The replacement entries provide the same functionality by explicitly enumerating the paths implied by the embedded wildcards. It is generally true that replacing with enumerated entries will cause the protections table or view to increase in size. The Perforce Server usually tolerates well the increase in size, though a larger protections table is tolerated better than a larger client view. The overall result can be a significant performance improvement when the entries that use paths with embedded wildcards are replaced with enumerated entries in one or both of the protections table and the view joined with the protections table.

The following graphics show an example of the effect of replacing entries that use paths with embedded wildcards with their functionally equivalent (for the example dataset) enumerated entries. For each combination, the graphics show the user and system CPU time and the db.revhx scans and positions for a "sync" command's compute phase returning 7,000 files:





The example "sync" command's compute phase used over twelve seconds of CPU time and scanned 700,000 db.revhx records when entries using paths with embedded wildcards were present in both the protections table and the client view. When the entries using paths with embedded wildcards were replaced with their functionally equivalent enumerated entries in either the protections table or the client view, the compute phase used approximately two-tenths of a second of CPU time and scanned at most one-tenth of the db.revhx records scanned when entries using paths with embedded wildcards were present in both the protections table and the client view. The best performance resulted when the protections table and the client view were both void of entries using paths with embedded wildcards. When using only enumerated entries, negligible CPU time was used and a minimal number of db.revhx records were scanned.

The ten protections table entries using paths with embedded wildcards for this example were:

```
write group dev * //depot/main/.../00/...
write group dev * //depot/main/.../01/...
...
write group dev * //depot/main/.../08/...
write group dev * //depot/main/.../09/...
```

The one thousand protections table enumerated entries for this example were:

```
write group dev * //depot/main/00/00/...
write group dev * //depot/main/00/01/...
...
write group dev * //depot/main/00/08/...
write group dev * //depot/main/00/09/...
```

© 2007 Perforce Software

```
write group dev * //depot/main/01/00/...
write group dev * //depot/main/01/01/...
...
write group dev * //depot/main/98/08/...
write group dev * //depot/main/98/09/...
write group dev * //depot/main/99/00/...
write group dev * //depot/main/99/01/...
...
write group dev * //depot/main/99/08/...
write group dev * //depot/main/99/09/...
```

The one client view entry using paths with embedded wildcards for this example was:

```
//depot/main/.../00/... //dev-client/depot/main/.../00/...
```

The one hundred client view enumerated entries for this example were:

```
//depot/main/00/00/... //dev-client/depot/main/00/00/...
//depot/main/01/00/... //dev-client/depot/main/01/00/...
...
//depot/main/98/00/... //dev-client/depot/main/98/00/...
//depot/main/99/00/... //dev-client/depot/main/99/00/...
```

Since the protections table is typically maintained by a limited number of users, it can be more easily controlled than views such as client views, which are typically maintained by every user. Ensuring that the use of paths with embedded wildcards is minimized in the protections table reduces the possibility of degraded performance resulting from joining the protections table with a view that uses paths with embedded wildcards.

6 Summary

Several areas of Perforce Server performance were detailed, and several misconceptions identified. In the area of metadata locking, the Perforce Server takes either shared or exclusive locks on only a subset of the db.* files needed to process a command. Locks are taken and released as needed while processing the command. On the topic of filesystem caching, the Perforce Server can realize a significant performance improvement when adequate physical memory is available for the operating system's use as a large filesystem cache. The filesystem cache only contains recently used portions of many of the db.* files, rather than the entirety of just a few of the db.* files. Exclusionary entries might make the protections table or a client view easier to understand, but they can require additional scans by the Perforce Server. For best performance, only inclusionary entries should be used. Joining the protections table with a client view when both use entries with embedded wildcards might degrade performance. Within the protections table, the use of entries with embedded wildcards should be minimized, which might require restructuring the repository or replacing with enumerated entries.

© 2007 Perforce Software