# Still All on One Server: Perforce at Scale

Dan Bloch, Google Inc.

May 21, 2011

## Abstract

Google runs the busiest single Perforce server on the planet, and one of the largest repositories in any source control system. From this high-water mark this paper looks at server performance and other issues of scale, with digressions into where we are, how we got here, and how we continue to stay one step ahead of our users.

## Perforce at Google

Google's main Perforce server supports over twelve thousand users, has more than a terabyte of metadata, and performs 11-12 million commands on an average day. The server runs on a 16-core machine with 256 GB of memory, running Linux. The metadata is on solid state disk, the depot files are on network-attached storage, and the logs and journal are on local RAID.

This server instance has been in operation for more than eleven years, and just passed changelist 20,000,000 (of which slightly under ten million are actual changelists). Our largest client has six million files; our largest changelists have in the low hundreds of thousands of files.

We have a single large depot with almost all of Google's projects on it. This aids agile development and is much loved by our users, since it allows almost anyone to easily view almost any code, allows projects to share code, and allows engineers to move freely from project to project. Documentation and data is stored on the server as well as code.

Our usage encompass almost all possible use patterns. More than half of it comes from programs and automated scripts. Of our interactive use, most is from the command line, but a sizeable minority of users do use the various GUIs and the P4Eclipse plug-in. A distinctive feature of the Google environment, which is extremely popular but quite expensive in terms of server load, is our code review tool, "Mondrian", which provides a dashboard from which users can browse and review changelists.

The title of this paper notwithstanding, Google does have about ten smaller servers, with metadata sizes ranging from 1 GB to 25 GB. The load on all of the smaller servers together is less than 20% of the load of the main server and for the most part they run themselves. This paper focuses exclusively on the main server.

## Performance

As advertised, Perforce is a fast version control system, and for smaller sites, it just works out of the box. This is amply demonstrated by Google's smaller servers (which really aren't that small), which run with almost no special attention from us. At larger sites, administrators typically do need to take an interest in performance issues, and for Google, which is at the edge of Perforce's envelope, performance is a major concern.

## Database Locking

In order to understand Perforce performance, one needs to understand the effect of database locking on concurrency. The issue is that Perforce metadata is stored in about forty database files in the P4ROOT directory on the server machine (with names like “db.have” “db.integed”, etc.). The p4d processes invoked on the server for each command takes locks on these database files at certain points in their execution. While these locks are held, some or all other commands (depending on whether the lock is a read lock or a write lock) are prevented from running until the lock is dropped. This reduces concurrency, e.g., in the extreme case it’s possible to have a 16-CPU machine with plenty of excess capacity, but only one process can run until its locks are dropped.

For large Perforce sites, this is the single biggest factor affecting performance. For administrators of small sites, it’s still worth understanding that performance issues are often caused by a few users or a few commands, and if you can prevent these you can improve performance without making any other changes.

At Google, through a number of initiatives we went from seeing the server blocked routinely for up to ten minutes at a time in 2005 to gaps of thirty seconds to a minute in 2010. With escalating user expectations, this is no longer acceptable either, and we’re working to reduce it still further. Note that achieving these gains requires ongoing monitoring, working closely with users, and is never solved for all time. As noted below, many of the performance measures we’ve taken reflect back on this issue.

This is discussed in much more depth in my previous presentations, “Performance and Database Locking at Large Perforce Sites” (2006) and “Life on the Edge: Monitoring and Running A Very Large Perforce Installation” (2007).

## Performance Resources

Performance is a concern of any Perforce administrator. Performance work can be approached from a standpoint of resources or of usage. Resources, as with any software applications, consist of CPU, memory, disk I/O, and network bandwidth.

- **CPU** – CPU is generally not an issue for Perforce servers. Google is an exception in this regard, due mainly to an overly complex protect table, so we need to monitor our CPU usage for our capacity planning.

  A more subtle problem associated with CPU capacity is that if CPU usage is maxed out even for a short length of time, it can cause TCP connection requests to the Perforce server to time out.

- **Memory** – Memory is important for a Perforce server, but, unlike CPU, there’s a point beyond which more memory doesn’t help. More doesn’t hurt (it’s used for disk caching), but it may be an unnecessary expense.

  There’s a myth that memory should be large enough to hold the whole Perforce database, but this isn’t true. Unused database rows are never paged into memory.

  In general, the goal is to avoid paging. Determining the amount of memory you need isn’t always easy. Perforce recommends 1.5k/file (http://kb.perforce.com/article/5), but this doesn’t apply at very large sites, where much of the database consists of history which is no longer referenced, e.g., old branches.

  Google configures machines with the maximum memory allowed, but this is largely a holdover from painful experiences with insufficient memory in the past. Some ad hoc testing showed that our server, with a 1 TB database, could run with less than 64 GB. Note that Google has very fast disk I/O (see below), so your experience may vary. The value of testing on your own configuration can’t be overstated, and Perforce product support can be a valuable resource as well.

- **Disk I/O** – For a large site, Perforce performance is dependent on disk I/O more than on any other resource. As described above, concurrency is gated by the time that commands holding locks while accessing the database, and since the amount of time spent reading or writing to the database is a function of disk I/O performance, anything you can do to speed up disk I/O will speed up Perforce. This means that making sure your disk is correctly configured, keeping your database on local disk instead of on network disk, and striping your data across multiple disk spindles, e.g., with RAID 10, will all improve performance.

  Still more effective but more expensive is keeping your database on a solid state disk instead of on mechanical hard disk drives. Google uses a RAM-based solid state disk. For all but the absolute largest sites, Flash-memory-based products deliver almost the same performance at a much lower cost.

- **Network bandwidth** – typically not an issue, but it is possible to exceed bandwidth to Perforce clients, or to the depot if it's on network-attached storage. Google has seen this maxed out for brief periods.

# Other Performance Improvements

In addition to buying hardware, as discussed above, there are a number of other avenues for performance improvement, all of which need to be pursued.

## Perforce, Inc. Improvements

The easiest way to improve performance is to wait and let Perforce do it for you. Over the past five years Perforce has introduced a large number of performance improvements. The most appreciated from Google's point of view were the improvements to concurrency: a 40% decrease in time spent holding locks by submits of branched files in 2006.2, earlier dropping of locks on most tables by submit, integrate, and sync commands in 2007.3, and improvements to the locking algorithm for most commands to drop and retry locks instead of blocking in 2008.1. There have been many performance improvements in other areas as well—it pays to upgrade! (And to read the release notes.)

## Reducing Metadata

Unlike deleting depot files, if you can delete or otherwise reduce the amount of metadata this can improve performance, since commands which scan database tables have less work to do. At a minimum, it will speed up checkpoints. Also, metadata is typically stored on more expensive or more limited media than depot files, e.g., solid state disk or local RAID as opposed to NAS. Ways of reducing use of metadata that we've used include:

- **client and label cleanups** - db.have and db.label are the two database files whose size you can reduce by simple deletions. We do fairly aggressive client cleanups, of clients that haven't been used in three months (with email warnings to users, and the ability to restore clients if necessary), and annual label cleanups. As far as we know these are not performance improvements, since these tables are never scanned in their entirety. Also note that automatic labels, introduced in 2006.2 (to replace labels created with "p4 tag" or "p4 labelsync") takes up no space in db.label.

- **sparse clients** – Google uses a a file system integration built with the P4API which allows users to work with very small client specifications, containing only the directories in which they are editing files, while all other files come from the integration, which provides files on demand and caches them locally. This mechanism isn't publicly available, but many build systems allow the specification of a base path to provide files which aren't in the local source tree, e.g. the VPATH/vpath construct used in gnumake. This is less

flexible than the file system integration, since it will be synced to head instead of providing files at a different changelist level for each user. But if process can be put in place to avoid version skew, this would provide similar benefits.

- **sparse branches** – similar to sparse clients, sparse branches allow for small branch specifications with most of the branch content coming from the mainline. Perforce provides rudimentary support for these with overlay mappings as described in the Perforce Knowledge Base article on Sparse Branching. Google goes beyond this with a build integration which allows users to set the changelist level for the mainline files.

  The benefits of this are more sweeping than the benefits of sparse clients. Clients are eventually deleted, and the space they take up is recovered from the database. Branches are used to create and submit branch releases, which remain in the database forever. Sparse branches are a fairly recent innovation at Google, and in their absence, revisions from branches amount to ninety-five percent of our depot. We expect the use of sparse branches to slow this growth, and the next strategy to reverse it.

  Finally, large submits and large integrates hold database locks for a long time, so reducing the size of changelists improves concurrency.

- **obliterates** – "p4 obliterate" is typically used to clean up depot disk space (or to clean up the depot namespace, or to remove files submitted accidentally, or to remove sensitive files). We've found a new use for it, cleaning up metadata disk space.

  Historically, obliterate has been too slow for large scale obliterates on large servers, but Perforce 2009.2 introduced the undocumented "–a" (don't remove depot content) and "-h" (don't remove "have" records from clients) flags. These flags (primarily "–a") reduce the time it takes to do obliterates from hours to seconds.

  We've done one large-scale obliterate of all branches older than two years (after letting our users specify exceptions). This removed 11% of the file paths in the depot, saved about $100,000, and let us defer a hardware purchase by a quarter. We'll probably repeat this every year or two.

  It wasn't quite as straightforward as I've made it sound. Before you try it, be aware that:

  - It took a lot of negotiation with our release engineering group to convince them to allow this.
  - In order to get release engineering buy-in, I had to create a process to restore obliterated branches in case they were found to be needed after all. This turns out to be possible when obliterates are done with the "-a" flag, because the depot content still exists, but it is in no way supported by Perforce.
  - Due in large part to this, it took two weeks of work, including a lot of testing.

## Monitor and If Necessary, Kill User Commands

Automated monitoring is an essential part of maintaining any service, Perforce or otherwise, for conditions such as whether the server is still running, whether there's sufficient disk space, etc., and we have monitoring for all of these conditions. In simplest terms, the goal is that you should never be notified about a problem by one of your users.

Perforce has some special issues which it's necessary to guard against, since any user can affect the performance of the server as a whole. The Perforce product does provide some tools for this purpose: the MaxResults, MaxScanrows, and MaxLocktime resource limits, and Google uses these to good effect, but we've found monitoring necessary as well. In addition to notifying us by email of exceptional conditions, there are some conditions where the right solution is always to kill an offending command, and in these cases we've extended the definition of "monitoring" to include killing these processes.

**Note that it is possible to corrupt your database by killing commands. It's strongly encouraged to contact Perforce product support before doing this.** The "p4 monitor terminate" command is a safe alternative you can also experiment with.

We have two scripts that kill processes:

- **locks.pl** – this script notifies administrators of all command holding locks whenever the p4d process count goes above a certain threshold (currently 325 excluding idle processes). High process count is almost always due to a command holding locks and blocking other commands. This script will also kill commands if they are holding locks, have been running for more than a minute, and are readonly commands. This script is available in my public depot directory.
- **killer.pl** – this tails the server log and immediately kills known bad commands, e.g., "p4 files //depot/…" The same effect could also be achieved with the Perforce Broker (which didn't exist when the script was written).

We have two additional scripts which:

- notify administrators of commands using more than 3 GB of memory
- notify administrators of users using more than 90 CPU seconds in a minute. This indicates that the user has a script running multiple commands in parallel, which can be a very effective if unintentional denial of service attack, and is difficult to detect by other means.

This aspect of our operation was discussed in detail in my 2007 talk.

## Systems Which Offload Work From Perforce

Ultimately, your goal is to support your users, and if they can do some of their work without touching the Perforce server, you can support more of them.

- **replicas** –we maintain two general purpose replicas which are used by read-only processes, primarily continuous builds. As of Perforce 2010.2, replication is fully supported by Perforce, and is described in the Perforce Replication chapter of the **System Administrator's Guide**. Google's replication predates this Perforce support, so we use a system developed in-house.
- **other systems** – as mentioned above in the discussion of sparse clients, we use a file system integration that serves files which would otherwise require calls to "p4 sync". We also have some specialized database systems which contain much of the Perforce metadata and can be queried instead of the Perforce server. These are somewhat similar in concept to Perforce's P4toDB tool, but they predate it.

  One caveat about implementing services like this is that they may poll the Perforce server frequently and generate load of their own. But hopefully not as much as they save.

## Multiple Servers

Clearly, putting some projects on other Perforce servers reduces load on the main server, but there are limits to the benefits of this strategy. In general, the use of multiple servers should be determined by how their contents are used. Unrelated projects can go on different servers, but arbitrarily splitting servers to solve performance problems won't offset the additional cost in administration, inconvenience, and user education. The most successful of our server divisions are for entirely unrelated projects, e.g., new acquisitions. The least successful is one whose use is coupled with the main server, so users are often required to submit changes in parallel with changes on the main server.

# Other Issues of Scale

## Databases Files

A few of the forty or so database files are interesting enough to be worth mentioning.

- **db.have** contains the records of all the files synced in all clients. It's the largest database file at almost any Perforce site. At Google, due to a combination of sparse clients and many large branches, it's the second largest, behind db.integed.

  **db.label** contains one record for each file in each non-automatic label. It may be large or small, depending on the use of labels at your site.

  db.have and db.label are the only tables which you can easily shrink, by deleting clients or labels. Note that this reduction in size is only seen when the database is restored from a checkpoint.

- **db.integed** contains integration records, and is another very large table. It's unique among the database files in that its records typically aren't accessed in sequential order on the disk, so access is greatly improved by use of a solid state disk.

- **db.rev** contains one record for each revision of each file in the database. There are also **db.revhx, db.revcx, and db.revdx,** which are additional indexes for optimizations.

  The above are the large databases in a typical site.

- **db.working** – contains one record for each open file. db.working is scanned by "p4 opened -a", which can take a long time. This command is automatically killed at Google for that reason. More problematically, db.working is scanned by "p4 user -d".

- **db.monitor** – contains one record for each Perforce command, as seen in "p4 monitor show" output. This table gets scanned almost constantly, in order to execute "p4 monitor terminate" commands in a timely fashion. For this reason, any system-wide problem (disk, memory) may incorrectly appear to be a problem with db.monitor, just because it's accessed so frequently.

## Depot Disk Space

Disk space is not a performance consideration, but it's often a capacity planning consideration or a financial consideration. Ideally, one tracks the use of disk space and communicates with users, so it's never a problem, but in actuality one will sometimes want to clean up files in the depot.

There are a number of methods to clean up depot files:

- obliterating files
- stubbing out files (http://kb.perforce.com/article/72/making-large-repositories-smaller)
- moving files to less expensive storage, with archive depots and the "p4 archive" and "p4 restore" commands, as of Perforce 2010.2 (not to be confused with the unrelated +X filetype and "archive" triggers).
- replacing multiple copies of identical files with hard links (rare)

A last note about disk space is that Perforce has a "+S" filetype (typically seen as "binary+S"), which is designed to save space by keeping only the most recent version(s) of a file. However, it has a hidden problem. If a +S file is branched, the branched file is a real copy, not a lazy copy (because the original may not always be around). This means that if a file is branched many times, +S will make the file and its copies take up more space, not less. **The "+S" filetype should only be used for files which won't be branched.** (Or will be branched very rarely.)

## Checkpoints

There are several strategies for checkpoints, and Google has used all of them at one time or another. Normally, the server is unavailable during the checkpoint. We stopped doing this four or five years ago when checkpoints took about four hours, and switched to offline checkpoints. Offline checkpoints worked well but took a very long time, which was of some concern even though it didn't involve downtime. Finally we switched to doing checkpoints off of Logical Volume

Manager (LVM) snapshots, which we remain very happy with. See "Demystifying Perforce Backups and Near Real-Time Replication" (Richard Baum, 2008).

Additionally, we checkpoint our replicas at the same time as we checkpoint the main server, so we have multiple copies of each checkpoint for redundancy.

Finally, we optimize our checkpoints and restores by processing each database file individually. This is straightforward to do by creating a subdirectories and populating each with a link to a database file. This means, since we have lots of CPUs and lots of bandwidth, that we can checkpoint and restore multiple database files in parallel, so our checkpoints and restores take five and a half hours instead of the seventeen they would if we just used p4d –jd and –jr.

## High availability

With more users and a global workforce comes a demand for higher and higher availability. We average about an hour a month of downtime, including scheduled downtime for failovers and upgrades. A few practices are key to this:

- You need a hot standby, and a failover plan, and you need to test your failovers. We schedule one failover a quarter. See "Perforce Disaster Recovery at Google" (Rick Wright, 2009).
- You need a test server, and you should make every effort to have it look exactly the same as your real server. This lets you avoid the two extremes of running tests on your live server and launching things untested.
- Every outage should have a postmortem, with action items to prevent it from happening again.

## Administrative Load

When your team is managing ten servers and 10,000 users, you need to make every effort to reduce the load on the administrators or you will soon be overwhelmed. Again, a few key practices:

- If you manage multiple servers, make all of them look the same: same disk layout (we have directories /p4_1, /p4_2, … and any server instance can be active on any machine), same scripts to run checkpoints and other maintenance, same triggers as much as possible, etc.
- A specific case of the above: have the same user accounts on all your servers. In order to do this last, Perforce provides a feature called P4AUTH (formerly undocumented; now supported as of 2010.2) under which the user and group data, and to some extent the protect table, are shared by all your servers. This means that accounts only have to be created once, and users have the same password on all servers.
- Invest in automating tasks. Batch up operations, e.g., delete former users' accounts in batches. Write self-service mechanisms for common requests, e.g., we have a web application which lets new users create their accounts with the appropriate access.
- Licensing: For larger customers with frequent purchase history, Perforce can offer a semi-annual purchase scheme, which enables easier deployment of Perforce user licenses with less administrative overhead for both parties. This scheme doesn't change the cost structure, but license availability is immediate. Please contact Perforce for more details.
- Share all information. For example, we cc email to Perforce Product Support to a common mailing list, so that all the administrators can learn from all the tickets. Another example is that we use IRC to communicate while issues are in progress.
- Document everything. You may not be the one to do any given task the next time. Or you may have forgotten how by then.

## Closing Thoughts

What has allowed all of this is that we grew into it. A Perforce site of this size wouldn't have been possible five years ago. Google's administrators couldn't have driven it, and Perforce's software couldn't have supported it. Google has had the opportunity to grow along with Perforce, in a partnership which has also been of value to the Perforce community as a whole, as more and more sites grow larger and larger.

There is some pain in being a pioneer: one recurring theme is that Perforce now provides solutions to many problems which we've already spent time designing our own solutions for, but this is minor. It's been a privilege being involved in all of this progress, and I trust that our experiences can be of some benefit to those who come after us.

## References


**Performance and Database Locking at Large Perforce Sites** (Dan Bloch, 2006), http://www.perforce.com/perforce/conferences/eu/2006/presentations/Google.pdf

**Life on the Edge: Monitoring and Running A Very Large Perforce Installation** (Dan Bloch, 2007), http://www.perforce.com/perforce/conferences/us/2007/presentations/DBloch_Life_on_the_Edge2007_paper.pdf

**Demystifying Perforce Backups and Near Real-Time Replication** (Richard Baum, 2008), http://www.perforce.com/perforce/conferences/eu/2008/presentations/perforce_richard_baum_whitepaper.pdf

**Perforce Disaster Recovery at Google** (Rick Wright, 2009), http://www.perforce.com/perforce/conferences/us/2009/Presentations/Wright-Disaster_Recovery-paper.pdf

**Perforce Replication** (System Administrator's Guide), http://www.perforce.com/perforce/r10.2/manuals/p4sag/10_replication.html