Optimizing Git for handling database backups: findings and insights from another 6,000 hours of experimentation

JV Roig
Advanced Research & Consulting, Asia Pacific College, Philippines
E-mail: jvr@apc.edu.ph; jvroig@gmail.com

Abstract. This is the second in a series of papers detailing results from on-going investigations of using Git - a distributed source control system - as a database backup and disaster recovery tool. The first paper, published early 2017 after 2,000 hours of data processing and simulations, warns that the default configuration of Git is unsuitable for database backups, but also shares configuration tweaks that make it safe and more space-efficient than zipped .sql archives - as low as only 1% of the storage space needed by .sql archives. This second paper focuses on optimization, looking specifically to identify which configuration tweaks and factors affect the reliability, scalability and efficiency of a Git-based database backup and disaster recovery system. Critical insights regarding the reliability and scalability of a Git-based database backup system were found after 6,000 hours of experimentation, centered around the problem of configuring Git to be able to handle .sql files measured in gigabytes.

1. Introduction
This is the second in a series of papers that share the results of my on-going experiments in deploying Git – one of the most popular source control systems in use today – as a database backup and disaster recovery system. For most companies, there is an obvious benefit to be gained (if using Git proves to be reliable for database backup and disaster recovery): Being one of the most popular source control systems, Git know-how is likely already a skill that exists within a company’s technology team, so using the same tool for backups and disaster recovery means being able to leverage existing skillsets. Git is also free and open-source software, so it has the further advantage of being free (zero cost) to deploy within the organization for any use.

This paper shares the results of thousands of hours of experimentation aimed at finding out how to properly optimize Git – that is, tweak relevant or critical configuration settings of Git – in order to maximize the reliability and scalability of Git for use as a database backup and disaster recovery system.

2. Background and Related Literature
The first paper in this series (published early 2017) dealt with the initial investigation of feasibility: “Could Git be a reasonable, cheap option for managing database backups?” [1] The results found in this study was promising:
Git can be safely used for database backups as long as the configuration `windowMemory` and `threads` are set to avoid exhausting all RAM, and `bigFileThreshold` is set high enough to be able to delta-compress your biggest expected `.sql` file size.

- Delta compression used by Git resulted in >99% storage space savings compared to a collection of zipped `.sql` files.

There has been little published information about investigative experiments or results regarding the use of Git for database backups. This was true when I first started the experiments in late 2016 and had it published early 2017. This scarcity of data persists today. The interest in using Git for database backups, however, is common from developers of all kinds, as can be seen from company blogs, personal blogs, and even in popular developer-oriented forums like StackOverflow. [2] [3] [4] [5] Part of this is due to Git's lower storage space usage compared to traditional source control systems. [6][7]

While the current definitive text available in public literature for experimental data in using Git for database backups remains my original paper, it has a couple of limitations:

- It does not investigate factors that may affect the reliability and scalability of Git.
- The individual file sizes handled in those experiments were under 2GB, with most being under 1GB, as those were the available backup data on hand.

The follow-up experiments described in this paper fill those gaps. Whereas the first paper was based on 2,000 hours of processing and simulations, this second paper is based on an additional 6,000 hours.

### 3. Experiment Design and Methodology

6,000 hours of data processing and simulations were done using 4 research servers (specs detailed in Table 1) handling 1,000 zip files. These zip files contain `.sql` files that are 1.5 – 2.5 GB each. These `.sql` files are historical backup data from an actual production system that I manage. Each server processes the thousand `.sql` files using different combinations of different Git settings in order to figure out which settings – or combination of settings – yield the lowest backup footprint (lowest storage space needed for the resulting Git repository) and which settings affect the speed of checking out a specific snapshot (i.e, how fast it takes for Git to retrieve a specific SQL file from a specific backup date).

In general, the experiment design and methodology used has not diverged much from the experimental design described in the first paper. An automated script, with only minor modifications from the original script used in the first paper, carries out the same steps as before per experiment:

1. A Git repository is initialized inside the main experiment folder. The configuration file of Git is also automatically tweaked according to the desired settings being for this experiment.
2. A zip file is selected (in chronological order according to the timestamp in the filename) from a folder that contains all 1,000 zipped `.sql` files the experiment will process.
3. This zip file is copied inside the Git repository, any existing `.sql` files are deleted, the contents of the zip file are extracted, then the zip file is deleted. Only the new `.sql` files remain in the working directory of the Git repository after this step.
4. The changes are staged and committed into the Git repository.
5. Depending on the current experiment being run, the script will trigger Git’s garbage collection at a certain loop number. In some experiments, this may only be at the very last commit; in others, this may be every n commits.
6. The script loops back to step #2 until all 1,000 zip files have been processed and their contents committed to the Git repository.

Once the script finishes one entire run, the resulting Git repository has simulated what it would have looked like if the database backups were done using Git: it now contains every backup file, in the correct, chronological order. The 1,000 zip files represent 500 days of backups; the simulation done by the script takes only 1 or 2 days, depending on the specific settings of the experiment.
There is also a separate script that does automated checkouts, retrieving each commit one at a time. This tests how fast or slow retrieval becomes based on various Git settings used to create the repository.

The experiments are focused on testing the effects and contributions of the following Git settings:

- **threads** – how many threads should Git use for our particular use case?
- **bigFileThreshold** – this setting limits the biggest file size that Git will consider for delta compression. Given the larger file sizes in these new sets of experiments, how much should this setting be increased?
- **windowMemory** – this setting limits how much memory Git will use during packing. How far should this setting be increased without risking exhausting all RAM and swapping, effectively killing the commit process, and therefore the backup process?
- **depth** – how long should the delta chains be, and how does it affect space usage and retrieval times?
- **GCMOD** - not an internal Git setting itself, but an important configuration tweak for the entire backup process itself: how often to trigger Git’s garbage collection (the `git gc` command). The name comes from “garbage collection modulo”, the operation that computes whether the script should trigger `git gc` or not.

Each experiment is a combination of various values for the above settings. The raw results of each (processing time, final repository size, and checkout/retrieval speed) are logged by the script itself, which I then collected afterwards to process and summarize. The volume of experiments done is summarized in Table 2.

Table 1. Specifications of the research servers used. They are all identical to minimize variance.

| Server Name | CPU                  | RAM       | OS        |
|-------------|----------------------|-----------|-----------|
| Galaxy      | Intel i7-6700 (Skylake) | 32GB DDR4 | Fedora 25 |
| Constitution| Intel i7-6700 (Skylake) | 32GB DDR4 | Fedora 25 |
| Sovereign   | Intel i7-6700 (Skylake) | 32GB DDR4 | Fedora 25 |
| Intrepid    | Intel i7-6700 (Skylake) | 32GB DDR4 | Fedora 25 |

Table 2. Experimentation volume summary.

| Server Name | # of Commit Experiments | # of Checkout Experiments | Total Experiment Time (hours) |
|-------------|-------------------------|---------------------------|-------------------------------|
| Galaxy      | 88                      | 160                       | 2,570.14                      |
| Constitution| 80                      | 152                       | 2,506.92                      |
| Sovereign   | 32                      | 44                        | 618.87                        |
| Intrepid    | 17                      | 29                        | 367.93                        |
| TOTAL       | 217                     | 385                       | 6,083.86                      |

Table 3 shows a partial list of the experiments done in the “Galaxy” server. To make sure that results are consistent and replicable, most experiments were duplicated in a separate server. Almost the entire experiment list of “Galaxy”, for example, was also done in the “Constitution” server. The entirety of the work done by all servers, including the full list of experiment configurations, raw results, and the automated scripts themselves are available as supplementary information from the dedicated research website for this experiment: https://research.jvroig.com/git_research_2.

4. Analysis and Results

After 6,000 hours spread out over 600 total experiments, critical insights regarding the reliability and scalability of a Git-based backup system were found. By reliability, I refer to the characteristic of the backup process not failing (i.e., the .sql file is always successfully committed into the Git repository).
By *scalability*, I refer to the characteristic of being able to handle bigger backup sizes and more volume, and also retaining an acceptable retrieval speed.

**Table 3.** Partial list of experiments in “Galaxy” server to illustrate the exploration of different Git settings. The columns “GCMOD” to “BigFileThreshold” refer to configuration values of those Git settings for the experiment. These Git settings were described earlier in section 3.

| Exp # | GCMOD | Depth | Threads | WindowMemory | BigFileThreshold |
|-------|-------|-------|---------|--------------|------------------|
| 1     | 5     | 50    | 1       | 8G           | 6G               |
| 2     | 5     | 200   | 1       | 8G           | 6G               |
| 3     | 5     | 300   | 1       | 8G           | 6G               |
| 4     | 200   | 50    | 1       | 8G           | 6G               |
| 5     | 200   | 200   | 1       | 8G           | 6G               |
| 6     | 200   | 300   | 1       | 8G           | 6G               |
| 7     | 100   | 250   | 2       | 4G           | 2G               |
| 8     | 100   | 250   | 2       | 4G           | 6G               |
| 9     | 4     | 250   | 2       | 4G           | 2G               |
| 10    | 4     | 250   | 2       | 4G           | 6G               |

4.1. **Threads**

Increasing the *threads* setting showed no improvement in repository size. And since repository size was unaffected, the speed of retrieval was likewise unaffected. There is no upside in making Git use more than 1 CPU thread. There is a huge downside: Git’s memory consumption is multiplied by the number of threads being used. From a reliability perspective, this means using more than one thread will greatly increase the chances of Git using more memory and exhausting RAM. From a scalability perspective, this greatly limits the maximum file size you can successfully back up. With no upside but severe downsides in both scalability and reliability, Git’s *threads* setting should always be explicitly set to 1.

4.2. **BigFileThreshold and WindowMemory**

These two settings were found to be linked. Increasing only one or the other does nothing, if the other setting is a bottleneck (i.e., changing *bigFileThreshold* to 4G while retaining *windowMemory* at 2G is no better than having both at 2G). For maximizing reliability, the experimental results suggest that both settings should be set to twice the size of the largest .sql file size that you expect to handle. For example, if your backups routinely handle up to a maximum of 4G, they should both be set to 8G. A reminder of a critical finding from the original paper is warranted: the backup process will fail if Git exhausts the available RAM. For this reason, these settings assume that you have enough RAM in the backup machine that uses Git. Ideally, the machine should have more RAM installed than your highest setting (e.g., if *windowMemory* is set to 8G, it is recommended that your machine has more than 8GB installed, to take into account all other processes that also use RAM, including the OS itself.)

4.3. **Depth**

Git’s *depth* setting controls how long the delta chain is, i.e., how often until a full copy of a file is stored instead of a delta. The longer the delta chain (the higher the *depth* setting), the smaller the repository size becomes since fewer full copies of files are stored. However, this has a more significant effect in retrieval times, as having a longer delta chain means any commit to retrieve will take Git longer to process. Experimental results found that retrieval times increase much faster than the decrease in storage space. Using *depth=250* makes the repository size only ~75% compared to the
default of depth=50 (0.875GB vs 1.126GB); however, average checkout times are more than doubled, taking >250% of the original average checkout time (61.19 seconds vs 22.11 seconds). Even the slow checkout time is still in the ballpark of a minute, though, so it can be a reasonable trade-off to increase depth from its default value to maximize storage space. From a reliability and scalability perspective, there is no significant downside, so this is more of a specific optimization to fit the organization’s particular use case. Whether to do so would mostly depend on any service-level targets that may exist in your organization for recovery time commitments.

4.4. GCMOD
Increasing GCMOD means triggering Git’s garbage collection less frequently (the command *git gc*). In the experiments, a GCMOD of 5 means *git gc* is triggered every 5 commits, a GCMOD of 20 means the trigger is every 20 commits, and so on. Similar to the depth setting, the repository size also decreases as GCMOD increases. However, unlike depth, GCMOD does not make retrieval times lag as much. From GCMOD=20 to GCMOD=200, the repository size decreased by 35% (1.7GB down to 1.1GB), while retrieval times only got slower by 54% (14.36 seconds to 22.11 seconds).

More frequent garbage collection (lower GCMOD) does have the effect of lowering the peak size a Git repository will grow to, although the garbage-collected repository size (i.e., effective size right after garbage collection) is always bigger. Delaying garbage collection for a month, for example, will yield the smallest repository size right after garbage collection. However, the repository size will increase until then. This has no effect from a reliability perspective, but it may impact scalability. If you have a maximum limit to what the Git repository size should peak at, then you will have to adjust GCMOD to be more frequent (for example, if available space in your backup machine can only accommodate a 100GB repository, then you should trigger *git gc* before this limit is reached, so that the repository size shrinks back to ~1GB). In the absence of this concern, garbage collection can be set to a monthly schedule, such as triggering *git gc* through a cron job every beginning of the month.

4.5. Storage Efficiency
Similar to the original paper’s findings, Git shows extreme storage efficiency for .sql files. The data set is composed of 1,000 backup files, zipped, weighing in at 121GB. Unzipped to raw .sql files, these backup files weigh in at approximately 1.2 – 1.5 TB. Experimental results showed that Git can store all these in a repository that is less than 1GB (875MB), shown in Table 4 along with Git’s comparative % efficiency. Retrieval times average only a minute for this best space-optimized configuration. This best space-optimized result used the following settings: Threads=1, BigFileThreshold=6G, WindowMemory=8G, Depth=300 and GCMOD=200.

Depth=300 lowers file size by making the delta chain longer, while GCMOD=200 maximizes the number of commits before a delta chain is created. These two settings are the primary drivers of the vastly reduced file size (0.875 GB). The Threads setting, as noted previously, should always be 1. BigFileThreshold and WindowMemory only need to be set to be at least twice the expected largest individual file size. For this experiment, the largest file size is 2.5GB as noted in Section 3, so any setting that is at least 5G will suffice.

| Raw .SQL Files (GB) (estimate) | Zipped Archives (GB) | Git Repo Size (GB) | Git % vs Raw .SQL files | Git % vs Zipped Archives |
|--------------------------------|----------------------|--------------------|------------------------|-------------------------|
| 1,300                          | 121                  | 0.875              | 0.06731%               | 0.72314%                |

5. Conclusion and Applications
Small and medium enterprises can use insights in this paper to setup a free but extremely efficient database backup and recovery system using Git, since it is free and open-source software. This is compatible with practically all database products, as it only ever deals with dumped .sql files, and
dumped plain text .sql files are a near-universal-feature for almost all database products. Even large enterprises may be able to significantly reduce their backup costs by switching to a Git-based backup system from a proprietary, licensed system. For large enterprises that handle different database products (e.g., having SQL Server, Oracle, and Postgres in one data center), switching to a Git-based backup system can unify the database backup process and potentially lower operating costs.

This study has revealed that the biggest bottleneck for a Git-based backup & disaster recovery system isn’t the total database size. Rather, Git is only affected by the largest individual file size it ever needs to process – this is the basis for adjusting `bigFileThreshold` and `windowMemory`, as noted in section 4.2. To scale Git towards handling databases that are tens of gigabytes without demanding a server with extreme RAM requirements, one possible approach is to dump the database into separate .sql files per table, as opposed to one .sql file containing the entire database. Instead of being bottlenecked by the total size of the database, Git’s bottleneck would then shift to whatever the size of the largest table is. Testing this scenario is the recommended next step in this series of experiments, to further stretch the limits of Git scaling.

References

[1] Roig JV 2017 2,000 hours of git and databases: raising the bar for the minimum standard of database backups in the enterprise Proc. of the 17th Philippine Computing Science Congress pp 17-23
[2] Eisinger D 2009 Backup your databases in git Viget https://www.viget.com/articles/backup-your-database-in-git
[3] Kulbertis B 2011 Synchronizing a MySQL database with git and git hooks Ben Kulbertis http://ben.kulbertis.org/2011/10/synchronizing-a-mysql-database-with-git-and-git-hooks/
[4] Rafique J 2016 Automatic database backup using git (bitbucket/gitlab/github) Jabran Rafique http://jabran.me/articles/automatic-database-backup-using-git-hosting/
[5] Wobbily_col 2014 Is backing up a MySQL database in git a good idea? Software Engineering Stack Exchange http://softwareengineering.stackexchange.com/questions/241109/is-backing-up-a-mysql-database-in-git-a-good-idea
[6] Pearce S 2013 GitSvnComparison The Linux Kernel Organization Git Wiki https://git.wiki.kernel.org/index.php/GitSvnComparison
[7] Packard K 2007 Repository formats matter Keith Packards’s Blog https://keithp.com/blog/Repository_Formats_Matter/