Storage

2024-02-13

Storage: File Systems and IO

3 Machine Learning IO needs

There are 3 distinct IO needs in the ML workload:

1. You need to be able to feed the DataLoader fast - (super fast read, don't care about fast write) - requires sustainable load for hours and days
2. You need to be able to write checkpoints fast - (super fast write, fastish read as you will be resuming a few times) - requires burst writing - you want super fast to not block the training for long (unless you use some sort of cpu offloading to quickly unblock the training)
3. You need to be able to load and maintain your codebase - (medium speed for both reading and writing) - this also needs to be shared since you want all nodes to see the same codebase - as it happens only during the start or resume it'll happen infrequently

As you can see these 3 have very different requirements both on speed and sustainable load, and thus ideally you'd have 3 different filesystems, each optimized for the required use case.

If you have infinite funds, of course, get a single super-fast read, super-fast write, that can do that for days non-stop. But for most of us, this is not possible so getting 2 or 3 different types of partitions where you end up paying much less is a wiser choice.

Incoming suggestions from Ross Wightman to integrate:

- I'd try to separate volumes by workload, so keep the 'lots of small files', high churn like environments, code separate from bulk storage like datasets, checkpoints. Possibly even split those too since datasets are largely static and checkpoints are being rotated all the time
- When datasets are on network storage, just like bucket storage, they should consist of large files AND be read as large files (sequentially in large chunks, not mmapped!). Avoid seeking within datasets

- Setups like HF datasets can be deceiving, might look like one big file, but often being mmap'd and the IO read pattern is nuts, like 3-4x more iops than if you'd read them as individual files. Mmap loading can be turned off, but if that's the case, for a lot of datasets you move a problem into the DataLoader processes, requiring reading too much data into memory at once. Better awareness of tradeoffs for different use cases, and especially using Iterable streaming when appropriate.
- Note that once your datasets are optimally friendly for a large, distributed network filesystem, they can usually just be streamed from bucket storage in cloud systems that have that option. So better to move them off the network filesystem in that case.
- In a way, bucket storage like s3, via the interface limitations, enforces patterns that are reasonable for storage backends like this. It's ooh, it's mounted as a folder, I can do whatever I want (mmap files, write loads of little ones, delete them all, etc) that's the prob.
- One also cannot expect to treat a distributed filesystem like their local disk. If you separated volumes by workload you'd probably be able to utilize much higher % of the total storage. Don't mix high churn, small files with low churn large files.
- Also, note that once your datasets are optimally friendly for a large, distributed network filesystem, they can usually just be streamed from bucket storage in cloud systems that have that option. So better to move them off the network filesystem in that case.

Glossary

- NAS: Network Attached Storage
- SAN: Storage Area Network
- DAS: Direct-Attached storage
- NSD: Network Shared Disk
- OSS: Object storage server
- MDS: Metadata server
- MGS: Management server

Which file system to choose

Distributed Parallel File Systems are the fastest solutions

Distributed parallel file systems dramatically improve performance where hundreds to thousands of clients can access the shared storage simultaneously. They also help a lot with reducing hotspots (where some data pockets are accessed much more often than others).

The 2 excellent performing parallel file systems that I had experience with are:

- Lustre FS (Open Source) (Wiki)
- GPFS (IBM), recently renamed to IBM Storage Scale, and before that it was called IBM Spectrum Scale.

Both solutions have been around for 2+ decades. Both are POSIX-compliant. These are also not trivial to create - you have to setup a whole other cluster with multiple cpu-only VMs dedicated exclusively for those filesystems - only then you can mount those. As compared to weaker cloud-provided "built-in" solutions which take only a few screens of questions to answer in order to activate. And when creating the storage cluster there is a whole science to which VMs to choose for which functionality. For example, here is a Lustre guide on GCP.

case study: At JeanZay HPC (France) we were saving 2.3TB checkpoint in parallel on 384 processes in 40 secs! This is insanely fast - and it was GPFS over NVME drives.

NASA's cluster has a long long list of gotchas around using Lustre.

Some very useful pros of GFPS: - If you have a lot of small files, you can easily run out of inodes (`df -i` to check). GFPS 5.x never runs out of inodes, it dynamically creates more as needed - GPFS doesn't have the issue Lustre has where you can run out of disk space at 80% if one of the sub-disks got full and wasn't re-balanced in time - you can reliably use all 100% of the allocated storage. - GPFS doesn't use a central metadata server (or a cluster of those) which often becomes a bottleneck when dealing with small files. Just like data, metatada is handled by each node in the storage cluster. - GPFS comes with a native NSD client which is superior to the generic NFS client, but either can be used with it.

Other parallel file systems I don't yet have direct experience with:

- BeeGFS
- WekaIO
- DAOS (Distributed Asynchronous Object Storage) (Intel)
- NetApp

Most clouds provide at least one implementation of these, but not all. If your cloud provider doesn't provide at least one of these and they don't have a fast enough alternative to meet your needs you should reconsider.

OK'ish solutions

There are many OK'ish solutions offered by various cloud providers. Benchmark those seriously before you commit to any. Those are usually quite decent for handling large files and not so much for small files.

case study: As of this writing with GCP's Zonal FileStore over NFS solution `python -c "import torch"` takes 20 secs to execute, which is extremely slow! Once the files are cached it then takes ~2 secs. Installing a conda environment with a handful of prebuilt python packages can easily take 20-30 min! This solution we started with had been very painful and counter-productive to our work. This would impact anybody who has a lot of python packages and conda environments. But, of course, GCP provides much faster solutions as well.

Remote File System Clients

You will need to choose which client to use to connect the file system to your VM with.

The most common choice is: NFS - which has been around for 4 decades. It introduces an additional overhead and slows things down. So if there is a native client supported by your VM, you'd have an overall faster performance using it over NFS. For example, GPFS comes with an NSD client which is superior to NFS.

File Block size

If the file system you use uses a block size of 16mb, but the average size of your files is 16k, you will be using 1,000 times more disk space than the actual use. For example, you will see 100TB of disk space used when the actual disk space will be just 100MB.

footnote: On Linux the native file systems typically use a block size of 4k.

So often you might have 2 very different needs and require 2 different partitions optimized for different needs.

1. thousands to millions of tiny files - 4-8k block size
2. few large files - 2-16mb block size

case study: Python is so bad at having tens of thousand of tiny files that if you have many conda environments you are likely to run of inodes in some situations. At JeanZay HPC we had to ask for a special dedicated partition where we would install all conda environments because we kept running out of inodes on normal GPFS partitions. I think the problem is that those GPFS partitions were configured with 16MB block sizes, so this was not a suitable partition for 4KB-large files.

The good news is that modern solutions are starting to introduce a dynamic block size. For example, the most recent GPFS supports sub-blocks. So, for example, it's possible to configure GPFS with a block size of 2mb, with a sub-block of 8k, and then the tiny files get packed together as sub-blocks, thus not wasting too much disk space.

Cloud shared storage solutions

Here are shared file system storage solutions made available by various cloud providers:

- GCP
- Azure
- AWS

Local storage beats cloud storage

While cloud storage is cheaper the whole idea of fetching and processing your training data stream dynamically at training time is very problematic with a huge number of issues around it.

Same goes for dynamic offloading of checkpoints to the cloud.

It's so much better to have enough disk space locally for data loading.

For checkpointing there should be enough local disk space for saving a checkpoint in a fast and reliable way and then having a crontab job or a slurm job to offload it to the cloud. Always keep the last few checkpoints locally for a quick resume, should your job crash, as it'd be very expensive to wait to fetch the checkpoint from the cloud for a resume.

case study: we didn't have a choice and had to use cloud storage for dataloading during IDEFICS-80B training as we had barely any local storage and since it was multimodal data it was many TBs of data. We spent many weeks trying to make this solution robust and it sucked at the end. The biggest issue was that it was very difficult at the time to keep track of RNG state for the DataSampler because the solution we used, well, didn't bother to take care of it. So a lot of data that took a lot of time to create was wasted (not used) and a lot of data was repeated, so we didn't have a single epoch of unique data.

Beware that you're often being sold only 80% of the storage you pay for

There is a subtle problem with distributed shared storage used on compute nodes. Since most physical disks used to build the large file systems are only 0.3-2TB large, any of these physical disks can get full before the combined storage gets full. And thus they require constant rebalancing so that there will be no situation where one disk is 99% full and others are only 50% full. Since rebalancing is a costly operation, like most programming languages' garbage collection, it happens infrequently. And so if you run `df` and it reports 90% full, it's very likely that any of the programs can fail at any given time.

From talking to IO engineers, the accepted reality (that for some reason is not being communicated to customers) is that only about 80% of distributed large storage is reliable.

Which means that if you want to have 100TB of reliable cloud storage you actually need to buy 125TB of storage, since 80% of that will be 100TB. So you need to plan to pay 25% more than what you provisioned for your actual needs. I'm not sure why the customer should pay for the technology deficiency but that's how it is.

For example, GCP states that only 89% can be used reliably, albeit more than once the storage failed already at 83% for me there. Kudos to Google to even disclosing this as a known issue, albeit not at the point of where a person buys the storage. As in - we recommend you buy 12% more storage than you actually plan to use, since we can only reliably deliver 89% of it.

I also talked to Sycomp engineers who provide managed IBM Storage Scale (GPFS) solutions, and according to them GPFS doesn't have this issue and the whole 100% can be reliably used.

Also on some setups if you do backups via the cloud provider API (not directly on the filesystem), they might end up using the same partition, and, of course, consume the disk space, but when you run `df` it will not show the real disk usage - it may show usage not including the backups. So if your backups consume 50% of the partition.

Whatever storage solution you pick, ask the provider how much of the storage can be reliably used, so that there will be no surprises later.

Beware that on some cloud providers backups use the same partition they backup

This makes no sense to me but with some providers when you make a back up of a partition using their tools, the back up will use space on that same partition. And on some of those providers you won't even know this happened until you run out of disk space when you really used 30% of the partition you allocated. On those providers running `df` is pointless because it'll tell you the free disk space, but it won't include any back ups in it. So you have no idea what's going on.

If you start making a backup and suddenly everything fails because all processes fail to write but `df` reports 30% usage, you will now know why this happened. Snapshots too use the same partition.

So say you paid for a 100TB partition and you used up 95TB and now you want to back it up - well, you can't - where would it put 95TB of data if it has 5TB of data left even if it compresses it.

As I discover specific solution that have this unintuitive behavior I will add pointers to how you can see the actual disk usage: - GCP FileStore (but it doesn't work for Basic Tier)

Don't forget the checksums

When you sync data to and from the cloud make sure to research whether the tool you use checks the checksums, otherwise you may end up with corrupt during transmission data. Some tools do it automatically, others you have to enable this feature (since it usually comes at additional compute cost and transmission slowdown). Better slow, but safe.

These are typically MD5 and SHA256 checksums. Usually MD5 is sufficient if your environment is safe, but if you want the additional security do SHA256 checksums.

Concepts

Here are a few key storage-related concepts that you likely need to be familiar with:

Queue Depth

Queue depth (or **IO depth**) is the number of IO requests that can be queued at one time on a storage device controller. If more IO requests than the controller can queue are being sent the OS will usually put those into its own queue.

On Linux the local block devices' queue depth is usually pre-configured by the kernel. For example, if you want to check the max queue depth set for `/dev/sda` you can `cat /sys/block/sda/queue/nr_requests`. To see the current queue depth of a local device run `iostat -x` and watch for `aqu-sz` column. (`apt install sysstat` to get `iostat`.)

Typically the more IO requests get buffered the bigger the latency will be, and the better the throughput will be. This is because if a request can't be acted upon immediately it'll prolong the response time as it has to wait before being served. But having multiple requests awaiting to be served in a device's queue would typically speed up the total throughput as there is less waiting time between issuing individual requests.

Direct vs Buffered IO

Direct IO refers to IO that bypasses the operating system's caching buffers. This corresponds to `O_DIRECT` flag in `open(2)` system call.

The opposite is the **buffered** IO, which is usually the default way most applications do IO since caching typically makes things faster.

When we run an IO benchmark it's critical to turn the caching/buffering off, because otherwise the benchmark's results will most likely be invalid. You normally won't be reading or writing the same file hundreds of times in a row. Hence most likely you'd want to turn the direct mode on in the benchmark's flags if it provides such.

In certain situation opening files with `O_DIRECT` may actually help to overcome delays. For example, if the training program logs to a log file (especially on a slow shared file system), you might not be able to see the logs for many seconds if both the application and the file system buffering are in the way. Opening the log file with `O_DIRECT` by the writer typically helps to get the reader see the logged lines much sooner.

Synchronous vs asynchronous IO

In synchronous IO the client submits an IO request and wait for it to be finished before submitting the next IO request to the same target device.

In asynchronous IO the client may submit multiple IO requests one after another without waiting for any to finish first. This requires that the target device can queue up multiple IO requests.

Sequential vs Random access IO

Sequential access IO is when you read blocks of data one by one sequentially (think a movie). Here are some examples: - reading or writing a model's checkpoint file all at once - loading a python program - installing a package

Random access IO is when you're accessing part of a file at random. Here are some examples: - database querying - reading samples from a pre-processed dataset in a random fashion - moving around a file using `seek`

Benchmarks

Time is money both in terms of a developer's time and model's training time, so it's crucial that storage IO isn't a bottleneck in your human and compute workflows.

In the following sections we will discuss various approaches to figuring out whether the proposed storage solution satisfies your work needs.

Metrics

The three main storage IO metrics one typically cares for are:

1. Throughput or Bandwidth (bytes per second - can be MBps, GBps, etc.)
2. IOPS (Input/output operations per second that a system can perform
3. Latency (msecs or usecs)

- *IOPS* measures how many input and/or output operations a given storage device or a cluster can perform per second. Typically read and write IOPS won't be the same. And for many systems it'll also depend on whether the operation is sequential or random. So a storage system will have 4 different IOPS rates:

1. IOPS of random reads
2. IOPS of random writes
3. IOPS of sequential reads
4. IOPS of sequential writes

- *Throughput* refers to how much data can be processed per second.

IOPS vs. Throughput

- when you deal with small files high IOPS is important.
- when you deal with large files high throughput is important.

IOPS correlates to Throughput via block size: `Throughput = IOPS * block_size`

Thus given a fixed IOPS - the larger the block size that the system can read or write the bigger the throughput will be.

And since there are 4 IOPS categories, correspondingly there are 4 throughput values to match.

Latency: is the delay between the moment the instruction to transfer data is issued and when the response to that instruction arrives.

Typically the more distance (switches, relays, actual distance) the packet has to travel the bigger the latency will be.

So if you have a local NVME drive your read or write latency will be much shorter as compared to reading or writing to a storage device that is located on another continent.

fio

fio - Flexible I/O tester is a commonly used IO benchmarking tool, which is relatively easy to operate. It has many options which allow you to emulate pretty much any type of a load and it provides a very detailed performance report.

First install `fio` with `apt install fio` or however your package manager does it.

Here is an example of a read benchmark:

```
base_path=/path/to/partition/
fio --ioengine=libaio --filesize=16k --ramp_time=2s --time_based --runtime=3m --numjobs=16 \
--direct=1 --verify=0 --randrepeat=0 --group_reporting --unlink=1 --directory=$base_path  \
--name=read-test --blocksize=4k --iodepth=64 --readwrite=read
```

Here 16 concurrent read threads will run for 3 minutes. The benchmark uses a block size of 4k (typical for most OSes) with the file size of 16k (a common size of most Python files) in a sequential reading style using non-buffered IO. So this particular set of flags will create a good benchmark to show how fast you can import Python modules on 16 concurrent processes.

case study: on one NFS setup we had `python -c "import torch"` taking 20 seconds the first time it was run, which is about 20x slower than the same test on a normal NVME drive. Granted once the files were cached the loading was much faster but it made for a very painful development process since everything was slow.

good read: Fio Output Explained - it's an oldie but is still a goodie - if you have a more up-to-date write up please send me a link or a PR.

Important: if you don't use the `--unlink=1` flag make sure to delete `fio`'s work files between different benchmarks - not doing so can lead to seriously wrong reports as `fio` will reuse files it prepared for a different benchmark which must not be re-used if the benchmark parameters have changed. Apparently this reuse is an `fio` feature, but to me it's a bug since I didn't know this nuance and got a whole lot of invalid reports because of it and it took awhile to realize they were wrong.

Going back to the benchmark - the parameters will need to change to fit the type of the IO operation you care to be fast - is it doing a lot of pip installs or writing a checkpoint on 512 processes, or doing a random read from a parquet file - each benchmark will have to be adapted to measure the right thing.

At the beginning I was manually fishing out the bits I was after, so I automated it resulting in fio-scan benchmark that will run a pair of read/write benchmarks on 16KB, 1MB and 1GB file sizes each using a fixed 4k block size (6 benchmarks in total). It uses a helper fio-json-extract.py to parse the log files and pull out the average latency, bandwidth and iops and report them in a nicely formatted markdown table.

Here is how to run it:

```
git clone https://github.com/stas00/ml-engineering/
cd ml-engineering
cd storage

path_to_test=/path/to/partition/to/test
./fio-scan $path_to_test
```

Adapt `path_to_test` to point to the partition path you want to benchmark.

note: the log parser uses python3. if `fio-scan` fails it's most likely because you run it on a system with python2 installed by default. It expects `python`

`--version` to be some python 3.x version. You can edit `fio-scan` to point to the right `python`.

Here is an example of this IO scan on my Samsung SSD 980 PRO 2TB NVME drive (summary):

- filesize=16k read

lat msec	bw MBps	IOPS	jobs
4.0	1006.3	257614	16

- filesize=16k write

lat msec	bw MBps	IOPS	jobs
3.2	1239.1	317200	16

- filesize=1m read

lat msec	bw MBps	IOPS	jobs
1.7	2400.1	614419	16

- filesize=1m write

lat msec	bw MBps	IOPS	jobs
2.1	1940.5	496765	16

- filesize=1g read

lat msec	bw MBps	IOPS	jobs
1.4	2762.0	707062	16

- filesize=1g write

lat msec	bw MBps	IOPS	jobs
2.1	1943.9	497638	16

As you can see as of this writing this is a pretty fast NVMe drive if you want to use it as a base-line against, say, a network shared file system.

Poor man's storage IO benchmark

Besides properly designed performance benchmarks which give you some numbers that you may or may not be able to appreciate there is a perception benchmark, and that is how does a certain functionality or a service feel. For example, when going to a website, does it feel like it's taking too long to load a webpage? or when going to a video service, does it take too long for the video to start playing and does it stop every few seconds to buffer the stream?

So with file system the questions are very simple - does it feel that it takes too long to install or launch a program? Since a lot of us live in the Python world, python is known to have thousands of tiny files which are usually installed into a virtual environment, with conda being the choice of many as of this writing.

In one of the environments we have noticed that our developers' productivity was really bad on a shared filesystem because it was taking up to 30min to install a conda environment with various packages needed for using a certain ML-training framework, and we also noticed that `python -c "import torch'` could take more than 20 seconds. This is about 5-10x slower than a fast local NVME-based filesystem would deliver. Obviously, this is bad. So I devised a perception test using `time` to measure the common activities. That way we could quickly tell if the proposed shared file system solution that we contemplated to switch to were significantly better. We didn't want a solution that was 2x faster, we wanted a solution that was 10x better, because having an expensive developer wait for proverbial paint to dry is not a good thing for a business.

So here is the poor man's benchmark that we used, so this is just an example. Surely if you think about the workflow of your developers you would quickly identify where things are slow and devise yours best fitting your needs.

note: To have a baseline to compare to do these timing tests on a recently manufactured local NVME. This way you know what the ceiling is, but with beware that many shared file systems won't be able to match that.

Step 1. Install conda onto the shared file system you want to test if it's not there already.

```
export target_partition_path=/mnt/weka  # edit me!!!
mkdir -p $target_partition_path/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O $target_partit
bash $target_partition_path/miniconda3/miniconda.sh -b -u -p $target_partition_path/minicond
rm -rf $target_partition_path/miniconda3/miniconda.sh
$target_partition_path/miniconda3/bin/conda init bash
bash
```

notes: - adapt `target_partition_path` and the miniconda download link if you aren't on the x86 platform. - at the end we launch a new `bash` shell for conda setup to take an effect, you might need to tweak things further if you're not a `bash` user - I trust you will know what to do.

Step 2. Measure conda install time (write test)

Time the creation of a new conda environment:

```
time conda create -y -n install-test python=3.9
```

```
real    0m29.657s
user    0m9.141s
sys     0m2.861s
```

Time the installation of some heavy pip packages:

```
conda deactivate
conda activate install-test
time pip install torch torchvision torchaudio
```

```
real    2m10.355s
user    0m50.547s
sys     0m12.144s
```

Please note that this test is somewhat skewed since it also includes the packages download in it and depending on your incoming network speed it could be super fast or super slow and could impact the outcome. But once the downloaded packages are cached, in the case of conda they are also untarred, so if you try to install the packages the 2nd time the benchmark will no longer be fair as on a slow shared file system the untarring could be very slow and we want to catch that.

I don't worry about it because usually when the file system is very slow usually you can tell it's very slow even if the downloads are slow, you just watch the progress and you can just tell.

If you do want to make this benchmark precise, you probably could keep the pre-downloaded conda packages and just deleting their untar'ed dirs:

```
find $target_partition_path/miniconda3/pkgs -mindepth 1 -type d -exec rm -rf {} +
```

in the case of `pip` it doesn't untar anything, but just caches the wheels it downloaded, so the `time pip install` benchmark can definitely be more precise if you run it the 2nd time (the first time it's downloaded, cached and installed, the second time it's installed from cache. So you could do:

```
conda create -y -n install-test python=3.9
conda activate install-test
pip install torch torchvision torchaudio
conda create -y -n install-test2 python=3.9
conda activate install-test2
time pip install torch torchvision torchaudio
```

As you can see here we time only the 2nd time we install the pip packages.

Step 3. Measure loading time after flushing the memory and file system caches (read test)

```
sudo sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
time python -c "import torch"
```

As you can see before we do the measurement we have to tell the OS to flush its memory and file system caches.

If you don't have `sudo` access you can skip the command involving `sudo`, also sometimes the system is setup to work w/o `sudo`. If you can't run the syncing and flushing of the file system caches you will just get incorrect results as the benchmark will be measuring the time to load already cached file system objects. To overcome this either ask your sysadmin to do it for you or simply come back in the morning while hopefully your file system caches other things and evicts the python packages, and then repeat the python one liner then with the hope those files are no longer in the cache.

Here is how to see the caching effect:

```
$ time python -c "import torch"

real    0m5.404s
user    0m1.761s
sys     0m0.751s

$ time python -c "import torch"

real    0m1.977s
user    0m1.623s
sys     0m0.519s

$ sudo sync
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
$ time python -c "import torch"

real    0m5.698s
user    0m1.712s
sys     0m0.734s
```

You can see that the first time it wasn't cached and took ~3x longer, then when I run it the second time. And then I told the system to flush memory and file system caches and you can see it was 3x longer again.

I think it might be a good idea to do the memory and file system caching in the write tests again, since even there caching will make the benchmark appear faster than what it would be like in the real world where a new package is installed for the first time.

other tools

-
- HPC IO Benchmark Repository (`mdtest` has been merged into `ior` in 2017)
- DLIO

XXX: expand on how these are used when I get a chance to try those

Published benchmarks

Here are some published IO benchmarks:

- MLPerf via MLCommons publishes various hardware benchmarks that measure training, inference, storage and other tasks' performance. For example, here is the most recent as of this writing storage v0.5 results. Though I find the results are very difficult to make sense of - too many columns and no control whatsoever by the user, and each test uses different parameters - so how do you compare things.

Then various benchmarks that you can run yourself:

Why pay for more storage when you can easily clean it up instead

Talking to a few storage providers I understood that many companies don't bother cleaning up and just keep on buying more and more storage. If you're not that company and want to keep things tidy in the following sections I will share how to easily prune various caches that many of us in the Python/Pytorch ecosphere use (and a lot of those will apply to other ecospheres).

HuggingFace Hub caches

The very popular HuggingFace Hub makes it super easy to download models and datasets and cache them locally. What you might not be aware of is that whenever a new revision of the model or a dataset is released, the old revisions remain on your disk - so over time you are likely to have a lot of dead weight.

The cached files are usually found at `~/.cache/huggingface` but it's possible to override those with `HF_HOME` environment variable and place them elsewhere if your `/home/` doesn't have space for huge files. (and in the past those were `HUGGINGFACE_HUB_CACHE` and `TRANSFORMERS_CACHE` and some others).

The other solution that requires no mucking with environment variables, which requires you to remember to set them, is to symlink your cache to another partition. You could do it for all of your caches:

```
mkdir -p ~/.cache
```

```
mv ~/.cache /some/path/
ln -s /some/path/.cache ~/.cache
```

or just for HF hub caches:

```
mkdir -p ~/.cache/huggingface
mv ~/.cache/huggingface /some/path/
ln -s /some/path/cache/huggingface ~/.cache/cache/huggingface
```

The `mkdir` calls are there in case you have haven't used the caches yet, so they weren't there and they ensure the above code won't fail.

Now that you know where the caches are, you could, of course, nuke the whole cache every so often, but if these are huge models and datasets, and especially if there was some preprocessing done for the latter - you really won't want to repeat those time consuming tasks again and again. So I will teach you how to use special tools provided by HuggingFace to do the cleanup.

The way revisions work on the HF hub is by pointing `main` to the latest revision of the files while keeping the old revisions around should anyone want to use the older revision for some reason. Chance are very high you always want the latest revision, and so here is how to delete all old revisions and only keeping `main` in a few quick steps without tedious manual editing.

In terminal A:

```
$ pip install huggingface_hub["cli"] -U
$ huggingface-cli delete-cache --disable-tui
File to edit: /tmp/tmpundr7lky.txt
0 revisions selected counting for 0.0. Continue ? (y/N)
```

Do not answer the prompt and proceed with my instructions.

(note your tmp file will have a different path, so adjust it below)

In terminal B:

```
$ cp /tmp/tmpedbz00ox.txt cache.txt
$ perl -pi -e 's|^#(.*\(detached\).*)|$1|' cache.txt
$ cat cache.txt >>  /tmp/tmpundr7lky.txt
```

The perl one-liner uncommented out all lines that had `(detached)` in it - so can be wiped out. And then we pasted it back into the tmp file `huggingface-cli` expects to be edited.

Now go back to terminal A and hit: N, Y, Y, so it looks like:

```
0 revisions selected counting for 0.0. Continue ? (y/N) n
89 revisions selected counting for 211.7G. Continue ? (y/N) y
89 revisions selected counting for 211.7G. Confirm deletion ? (Y/n) y
```

Done.

If you messed up with the prompt answering you still have `cache.txt` file which you can feed again to the new tmp file it'll create when you run `huggingface-cli delete-cache --disable-tui` again.

attached as a snapshot as well as it's easier to read on twitter, but use the message to copy-n-paste from.

Please note that you can also use this tool to choose which models or datasets to delete completely. You just need to open `cache.txt` in your editor and remove the `#` in front of lines that contain `main` in it for models/datasets you want to be deleted for you. and then repeat the process explained above minus the `perl` one liner which you'd replace with manual editing.

Additionally you will find that HF `datasets` have a `~/.cache/huggingface/datasets/downloads` dir which often will contain a ton of leftovers from datasets downloads and their preprocessing, including various lock files. On one setup I found literally a few millions of files there. So here is how I clean those up:

```
sudo find ~/.cache/huggingface/datasets/downloads -type f -mtime +3 -exec rm {} \+
sudo find ~/.cache/huggingface/datasets/downloads -type d -empty -delete
```

The first command leaves files that are younger than 3 days in place, in case someone is in the process of download/processing things and we don't want to swipe the carpet from under their feet.

As usual you may need to adjust the paths if you placed your caches elsewhere.

Python package manager cleanups

conda and pip will pile up more and more files on your system over time. conda is the worst because it keeps the untarred files which consume an insane amount of inodes and make backups and scans slow. pip at least caches just the wheels (tarred files).

So you can safely nuke these dirs:

```
rm -rf ~/.cache/pip
rm -rf ~/anaconda3/pkgs/
```

Make sure edit the last command if your conda is installed elsewhere.

Share caches in group environments

If you have more than 2 people working on the same system, you really want to avoid each person having their own cache of `pip`, `conda`, HF models, datasets and possibly other things. It is very easy to get each user's setup to point to a shared cache.

For example, let's say you make `pip` and `conda` caches under `/data/cache` like so:

```
mkdir /data/cache/conda
mkdir /data/cache/pip
chmod a+rwx /data/cache/conda
chmod a+rwx /data/cache/pip
```

now you just need to symlink from each user's local cache to this shared cache:

```
mkdir -p ~/.cache

rm -rf ~/.cache/pip
ln -s /data/cache/pip ~/.cache/pip

rm -rf ~/.conda/pkgs
ln -s /data/cache/conda/pkgs ~/.conda/pkgs
```

note that we wiped out the existing caches, but you could also move them to the shared cache instead - whatever works, you will want to periodically nuke those anyway.

So now when `pip` or `conda` will try to reach the user caches they will get redirected to the shared cache. If you have 20 people in the group that's 20x less files - and this is very important because conda pkg files are untarred and take up a huge amount of inodes on the disk.

So the only issue with this approach is file permissions. If user A installs some packages, user B might not be able to read or write them.

If this is an isolated cluster where there are no malicious users you can simply ask everybody to use `umask 000` in their `~/.bashrc` or even configuring this setting system-wide via `/etc/profile` or `/etc/bash.bashrc` and different other shell config files if `bash` isn't your shell of choice.

Once `umask 000` is run, most files will be created with read/write perms so that all users can read/write each others files.

Of course, if you are using a sort of HPC, where many unrelated groups use the same cluster this won't work and then you would either use groups instead of making files read/write by all, with possibly `setgid` bit preset or using ACL . In any such environments there are always sysadmins so you can ask them how to setup a shared cache for your team and they will know what to do.

Additionally, recently some of these applications added tools to do the cleanup, e.g. for `conda` and `pip`:

```
conda clean --all -f -y
pip cache purge
```

General disk usage

Of course, sooner or later, your partition will get bigger and bigger, and you will probably want to understand where data is leaking. Typically you will need to find the users who contribute to the most of data consumption and ask them to do some cleanups.

So for example to find which users consume the most disk run:

```
sudo du -ahd1 /home/* | sort -rh
```

it will sort the data by the worst offenders. If you want to help them out you could go into their dirs and analyse the data a level deeper:

```
sudo du -ahd1 /home/*/* | sort -rh
```

or for a specific user `foo`:

```
sudo du -ahd1 /home/foo/* | sort -rh
```

You could also set disk usage quotas but usually this doesn't work too well, because depending on the workflows of your company some users need to generate a lot more data then others, so they shouldn't be punished for that with inability to do their work and have their job crash - which could have been run for many hours and all that work will be lost - so at the end of the day the company will be paying for the lost time.

Getting users to be aware of them using too much disk space can be a very difficult task.

Partition inodes limit

Also beware of inode usage, on some shared partitions on HPCs I have seen more than once cases where a job crashed not because there was no disk space left, but because the job used up the last inodes and the whole thing crashed.

To see inode usage, use `df -i`:

```
$ /bin/df -hi
Filesystem     Inodes IUsed IFree IUse% Mounted on
tmpfs             16M  1.9K   16M    1% /run
/dev/sda1         59M  4.1M   55M    7% /
```

`-h` formats huge numbers into human-readable strings.

So here you can see the the `/` partition is using 7% of the total possible inodes.

Depending on the type of filesystem in some cases it's possible to add more inodes whereas in other cases it's not possible.

So as part of your monitoring of disk space you also need to monitor inode usage as a critical resource.

`/tmp` on compute nodes

Normally compute nodes will use `/tmp/` for temp files. The problem is on most set ups `/tmp` resides on the tiny `/` filesystem of each node (often <100GB) and since `/tmp/` only gets reset on reboot, this doesn't get cleaned up between SLURM jobs and this leads to `/tmp` running out of space and so when you try to run something that let's say untars a file you're likely to run into:

```
OSError: [Errno 28] No space left on device
```

The solution is to set in your SLURM launcher script.

```
export TMPDIR=/scratch
```

Now, the slurm job will use a much larger `/scratch` instead of `/tmp`, so plenty of temp space to write too.

footnote: while `/scratch` is quite common - the mounted local SSD disk mount point could be named anything, e.g. `/localssd` - it should be easy to see the right path by running `df` on one of the compute nodes.

You can also arrange for the SLURM setup to automatically clean up such folders on job's termination.

How to find users who consume a lot of disk space

Do you have a problem when your team trains models and you constantly have to buy more storage because huge model checkpoints aren't being offloaded to bucket storage fast enough?

Here is a one-liner that will recursively analyze a path of your choice, find all the checkpoints, sum up their sizes and print the totals sorted by the biggest user, so that you could tell them to clean up their act :) Just edit `/mypath` to the actual path

```
find /mypath/ -regextype posix-egrep -regex ".*\.(pt|pth|ckpt|safetensors)$" | \
perl -nle 'chomp; ($uid,$size)=(stat($_))[4,7]; $x{$uid}+=$size;
END { map { printf qq[%-10s: %7.1fTB\n], (getpwuid($_))[0], $x{$_}/2**40 }
sort { $x{$b} <=> $x{$a} } keys %x }'
```

gives:

```
user_a    :     2.5TB
user_c    :     1.6TB
user_b   :      1.2TB
```

Of course, you can change the regex to match other patterns or you can remove it altogether to measure all files:

```
find /mypath/ | \
perl -nle 'chomp; ($uid,$size)=(stat($_))[4,7]; $x{$uid}+=$size;
END { map { printf qq[%-10s: %7.1fTB\n], (getpwuid($_))[0], $x{$_}/2**40 }
sort { $x{$b} <=> $x{$a} } keys %x }'
```

How to automatically delete old checkpoints

Continuing the item from above, if you want to automatically delete old checkpoints instead (e.g. those older than 30 days).

First try to ensure the candidates are indeed good to delete:

```
find /mypath/ -regextype posix-egrep -regex ".*\.(pt|pth|ckpt|safetensors)$" -mtime +30
```

and when you feel it's safe to delete, only then add `rm`

```
find /mypath/ -regextype posix-egrep -regex ".*\.(pt|pth|ckpt|safetensors)$" -mtime +30 -exe
```

Contributors

Ross Wightman