sparse-checkout

- Terminology ==
- Purpose of sparse-checkouts ==
  - (Behavior A) Users are only interested in the sparse portion of the repo
  - (Behavior A*) Users are only interested in the sparse portion of the repo that they have downloaded
  - (Behavior B) Users want a sparse working tree, but are working in a larger whole
  - (Behavior C) sparse-checkout is an implementational detail supporting a special VFS.
- Usecases of primary concern ==
  - (Behavior A*)
  - (Behavior C)
- Oversimplified mental models ==
- Desired behavior ==
- Behavior classes ==
- Subcommand-dependent defaults ==
- Sparse specification vs. sparsity patterns ==
- Implementation Questions ==
- Implementation Goals/Plans ==
- Known bugs ==
- Reference Emails ==

Table of contents:

- Terminology
- Purpose of sparse-checkouts
- Usecases of primary concern
- Oversimplified mental models ("Cliff Notes" for this document!)
- Desired behavior
- Behavior classes
- Subcommand-dependent defaults
- Sparse specification vs. sparsity patterns
- Implementation Questions

- Implementation Goals/Plans
- Known bugs
- Reference Emails

Terminology ==

`cone` `mode`
one of two modes for specifying the desired subset of files in a sparse-checkout. In cone-mode, the user specifies directories (getting both everything under that directory as well as everything in leading directories), while in non-cone mode, the user specifies gitignore-style patterns. Controlled by the --[no-]cone option to sparse-checkout init|set.

`SKIP_WORKTREE`
When tracked files do not match the sparse specification and are removed from the working tree, the file in the index is marked with a SKIP_WORKTREE bit. Note that if a tracked file has the SKIP_WORKTREE bit set but the file is later written by the user to the working tree anyway, the SKIP_WORKTREE bit will be cleared at the beginning of any subsequent Git operation.

Most sparse checkout users are unaware of this implementation detail, and the term should generally be avoided in user-facing descriptions and command flags. Unfortunately, prior to the `sparse-checkout` subcommand this low-level detail was exposed, and as of time of writing, is still exposed in various places.

`sparse-checkout`
a subcommand in git used to reduce the files present in the working tree to a subset of all tracked files. Also, the name of the file in the $GIT_DIR/info directory used to track the sparsity patterns corresponding to the user’s desired subset.

`sparse` `cone`
see cone mode

`sparse` `directory`
An entry in the index corresponding to a directory, which appears in the index instead of all the files under that directory that would normally appear. See also sparse-index. Something that can cause confusion is that the "sparse directory" does NOT match the sparse specification, i.e. the directory is NOT present in the working tree. May be renamed in the future (e.g. to "skipped directory").

`sparse` `index`
A special mode for sparse-checkout that also makes the index sparse by recording a directory entry in lieu of all the files underneath that directory (thus making that a "skipped directory" which unfortunately has also been called a "sparse directory"), and does this for potentially multiple directories. Controlled by the --[no-]sparse-index option to init|set|reapply.

`sparsity` `patterns`
patterns from $GIT_DIR/info/sparse-checkout used to define the set of files of interest. A warning: It is easy to over-use this term (or the shortened "patterns" term), for two reasons: (1) users in cone mode specify directories rather than patterns (their directories are transformed into patterns, but users may think you are talking about non-cone mode if you use the word "patterns"), and (2) the sparse specification might transiently differ in the working tree or index from the sparsity patterns (see "Sparse specification vs. sparsity patterns").

`sparse` `specification`
The set of paths in the user's area of focus. This is typically just the tracked files that match the sparsity patterns, but the sparse specification can temporarily differ and include additional files. (See also "Sparse specification vs. sparsity patterns")

- When working with history, the sparse specification is exactly the set of files matching the sparsity patterns.
- When interacting with the working tree, the sparse specification is the set of tracked files with a clear SKIP_WORKTREE bit or tracked files present in the working copy.
- When modifying or showing results from the index, the sparse specification is the set of files with a clear SKIP_WORKTREE bit or that differ in the index from HEAD.
- If working with the index and the working copy, the sparse specification is the union of the paths from above.

`vivifying`
When a command restores a tracked file to the working tree (and hopefully also clears the SKIP_WORKTREE bit in the index for that file), this is referred to as "vivifying" the file.

Purpose of sparse-checkouts

==

sparse-checkouts exist to allow users to work with a subset of their files.

You can think of sparse-checkouts as subdividing "tracked" files into two categories — a sparse subset, and all the rest. Implementationally, we mark "all the rest" in the index with a SKIP_WORKTREE bit and leave them out of the working tree. The SKIP_WORKTREE files are still tracked, just not present in the working tree.

In the past, sparse-checkouts were defined by "SKIP_WORKTREE means the file is missing from the working tree but pretend the file contents match HEAD". That was not only bogus (it actually meant the file missing from the working tree matched the index rather than HEAD), but it was also a low-level detail which only provided decent behavior for a few commands. There were a surprising number of ways in which that guiding principle gave command results that violated user

expectations, and as such was a bad mental model. However, it persisted for many years and may still be found in some corners of the code base.

Anyway, the idea of "working with a subset of files" is simple enough, but there are multiple different high-level usecases which affect how some Git subcommands should behave. Further, even if we only considered one of those usecases, sparse-checkouts can modify different subcommands in over a half dozen different ways. Let’s start by considering the high level usecases:

A)
Users are *only* interested in the sparse portion of the repo

A*)
Users are *only* interested in the sparse portion of the repo that they have downloaded so far

B)
Users want a sparse working tree, but are working in a larger whole

C)
sparse-checkout is a behind-the-scenes implementation detail allowing Git to work with a specially crafted in-house virtual file system; users are actually working with a "full" working tree that is lazily populated, and sparse-checkout helps with the lazy population piece.

It may be worth explaining each of these in a bit more detail:

(Behavior A) Users are *only* interested in the sparse portion of the repo

These folks might know there are other things in the repository, but don’t care. They are uninterested in other parts of the repository, and only want to know about changes within their area of interest. Showing them other files from history (e.g. from diff/log/grep/etc.) is a usability annoyance, potentially a huge one since other changes in history may dwarf the changes they are interested in.

Some of these users also arrive at this usecase from wanting to use partial clones together with sparse checkouts (in a way where they have downloaded blobs within the sparse specification) and do disconnected development. Not only do these users generally not care about other parts of the repository, but consider it a blocker for Git commands to try to operate on those. If commands attempt to access paths in history outside the sparsity specification, then the partial clone will attempt to download additional blobs on demand, fail, and then fail the user’s command. (This may

be unavoidable in some cases, e.g. when `git merge` has non-trivial changes to reconcile outside the sparse specification, but we should limit how often users are forced to connect to the network.)

Also, even for users using partial clones that do not mind being always connected to the network, the need to download blobs as side-effects of various other commands (such as the printed diffstat after a merge or pull) can lead to worries about local repository size growing unnecessarily[10].

(Behavior A*) Users are *only* interested in the sparse portion of the repo that they have downloaded so far (a variant on the first usecase)

This variant is driven by folks who using partial clones together with sparse checkouts and do disconnected development (so far sounding like a subset of behavior A users) and doing so on very large repositories. The reason for yet another variant is that downloading even just the blobs through history within their sparse specification may be too much, so they only download some. They would still like operations to succeed without network connectivity, though, so things like `git log -S${SEARCH_TERM} -p` or `git grep ${SEARCH_TERM} OLDREV` would need to be prepared to provide partial results that depend on what happens to have been downloaded.

This variant could be viewed as Behavior A with the sparse specification for history querying operations modified from "sparsity patterns" to "sparsity patterns limited to the blobs we have already downloaded".

(Behavior B) Users want a sparse working tree, but are working in a larger whole

Stolee described this usecase this way[11]:

"I’m also focused on users that know that they are a part of a larger whole. They know they are operating on a large repository but focus on what they need to contribute their part. I expect multiple "roles" to use very different, almost disjoint parts of the codebase. Some other "architect" users operate across the entire tree or hop between different sections of the codebase as necessary. In this situation, I’m wary of scoping too many features to the sparse-checkout definition, especially "git log," as it can be too confusing to have their view of the codebase depend on your "point of view."

People might also end up wanting behavior B due to complex inter-project dependencies. The initial attempts to use sparse-checkouts usually involve the directories you are directly interested in plus what those directories depend upon within your repository. But there’s a monkey wrench here: if you have integration tests, they invert the hierarchy: to run integration tests, you need not only what you are interested in and its in-tree dependencies, you also need everything that depends upon what you are interested in or that depends upon one of your dependencies…AND you need all the in-tree dependencies of that expanded group. That can easily change your sparse-checkout into a nearly dense one.

Naturally, that tends to kill the benefits of sparse-checkouts. There are a couple solutions to this conundrum: either avoid grabbing in-repo dependencies (maybe have built versions of your in-repo dependencies pulled from a CI cache somewhere), or say that users shouldn’t run integration tests directly and instead do it on the CI server when they submit a code review. Or do both. Regardless of whether you stub out your in-repo dependencies or stub out the things that depend upon you, there is certainly a reason to want to query and be aware of those other stubbed-out parts of the repository, particularly when the dependencies are complex or change relatively frequently. Thus, for such uses, sparse-checkouts can be used to limit what you directly build and modify, but these users do not necessarily want their sparse checkout paths to limit their queries of versions in history.

Some people may also be interested in behavior B over behavior A simply as a performance workaround: if they are using non-cone mode, then they have to deal with its inherent quadratic performance problems. In that mode, every operation that checks whether paths match the sparsity specification can be expensive. As such, these users may only be willing to pay for those expensive checks when interacting with the working copy, and may prefer getting "unrelated" results from their history queries over having slow commands.

(Behavior C) sparse-checkout is an implementational detail supporting a special VFS.

This usecase goes slightly against the traditional definition of sparse-checkout in that it actually tries to present a full or dense checkout to the user. However, this usecase utilizes the same underlying technical underpinnings in a new way which does provide some performance advantages to users. The basic idea is that a company can have an in-house Git-aware Virtual File System which pretends all files are present in the working tree, by intercepting all file system accesses and using those to fetch and write accessed files on demand via partial clones. The VFS uses sparse-checkout to prevent Git from writing or paying attention to many files, and manually updates the sparse checkout patterns itself based on user access and modification of files in the working tree. See commit ecc7c8841d ("repo_read_index: add config to expect files outside sparse patterns", 2022-02-25) and the link at [17] for a more detailed description of such a VFS.

The biggest difference here is that users are completely unaware that the sparse-checkout machinery is even in use. The sparse patterns are not specified by the user but rather are under the complete control of the VFS (and the patterns are updated frequently and dynamically by it). The user will perceive the checkout as dense, and commands should thus behave as if all files are present.

Usecases of primary concern

==

Most of the rest of this document will focus on Behavior A and Behavior B. Some notes about the other two cases and why we are not focusing on them:

(Behavior A*)

Supporting this usecase is estimated to be difficult and a lot of work. There are no plans to implement it currently, but it may be a potential future alternative. Knowing about the existence of additional alternatives may affect our choice of command line flags (e.g. if we need tri-state or quad-state flags rather than just binary flags), so it was still important to at least note.

Further, I believe the descriptions below for Behavior A are probably still valid for this usecase, with the only exception being that it redefines the sparse specification to restrict it to already-downloaded blobs. The hard part is in making commands capable of respecting that modified definition.

(Behavior C)

This usecase violates some of the early sparse-checkout documented assumptions (since files marked as SKIP_WORKTREE will be displayed to users as present in the working tree). That violation may mean various sparse-checkout related behaviors are not well suited to this usecase and we may need tweaks — to both documentation and code — to handle it. However, this usecase is also perhaps the simplest model to support in that everything behaves like a dense checkout with a few exceptions (e.g. branch checkouts and switches write fewer things, knowing the VFS will lazily write the rest on an as-needed basis).

Since there is no publicly available VFS-related code for folks to try, the number of folks who can test such a usecase is limited.

The primary reason to note the Behavior C usecase is that as we fix things to better support Behaviors A and B, there may be additional places where we need to make tweaks allowing folks in this usecase to get the original non-sparse treatment. For an example, see ecc7c8841d ("repo_read_index: add config to expect files outside sparse patterns", 2022-02-25). The secondary reason to note Behavior C, is so that folks taking advantage of Behavior C do not assume they are part of the Behavior B camp and propose patches that break things for the real Behavior B folks.

Oversimplified mental models ==

An oversimplification of the differences in the above behaviors is:

(Behavior A)
Restrict worktree and history operations to sparse specification

(Behavior B)
Restrict worktree operations to sparse specification; have any history operations work across all files

(Behavior C)
Do not restrict either worktree or history operations to the sparse specification…with the exception of branch checkouts or switches which avoid writing files that will match the index so they can later lazily be populated instead.

Desired behavior ==

As noted previously, despite the simple idea of just working with a subset of files, there are a range of different behavioral changes that need to be made to different subcommands to work well with

such a feature. See [1,2,3,4,5,6,7,8,9,10] for various examples. In particular, at [2], we saw that mere composition of other commands that individually worked correctly in a sparse-checkout context did not imply that the higher level command would work correctly; it sometimes requires further tweaks. So, understanding these differences can be beneficial.

- Commands behaving the same regardless of high-level use-case

- commands that only look at files within the sparsity specification

- diff (without --cached or REVISION arguments)
- grep (without --cached or REVISION arguments)
- diff-files

- commands that restore files to the working tree that match sparsity patterns, and remove unmodified files that don't match those patterns:

- switch
- checkout (the switch-like half)
- read-tree
- reset --hard

- commands that write conflicted files to the working tree, but otherwise will omit writing files to the working tree that do not match the sparsity patterns:

- merge
- rebase
- cherry-pick
- revert

- `am` and `apply` `--cached` should probably be in this section but are buggy (see the "Known bugs" section below)

```
The behavior for these commands somewhat depends upon the merge
strategy being used:
```

- `ort` behaves as described above
- `octopus` and `resolve` will always vivify any file changed in the merge relative to the first parent, which is rather suboptimal.

```
It is also important to note that these commands WILL update the index
outside the sparse specification relative to when the operation began,
BUT these commands often make a commit just before or after such that
by the end of the operation there is no change to the index outside the
sparse specification.  Of course, if the operation hits conflicts or
does not make a commit, then these operations clearly can modify the
```

```
index outside the sparse specification.

Finally, it is important to note that at least the first four of these
commands also try to remove differences between the sparse
specification and the sparsity patterns (much like the commands in the
previous section).
```

- commands that always ignore sparsity since commits must be full-tree

- archive
- bundle
- commit
- format-patch
- fast-export
- fast-import
- commit-tree

- commands that write any modified file to the working tree (conflicted or not, and whether those paths match sparsity patterns or not):

- stash
- apply (without `--index` or `--cached`)

- Commands that may slightly differ for behavior A vs. behavior B:

```
Commands in this category behave mostly the same between the two
behaviors, but may differ in verbosity and types of warning and error
messages.
```

- commands that make modifications to which files are tracked:

- add
- rm
- mv
- update-index

```
The fact that files can move between the 'tracked' and 'untracked'
categories means some commands will have to treat untracked files
differently.  But if we have to treat untracked files differently,
then additional commands may also need changes:
```

- status
- clean

```
In particular, `status` may need to report any untracked files outside
the sparsity specification as an erroneous condition (especially to
avoid the user trying to `git add` them, forcing `git add` to display
an error).

It's not clear to me exactly how (or even if) `clean` would change,
but it's the other command that also affects untracked files.

`update-index` may be slightly special.  Its --[no-]skip-worktree flag
may need to ignore the sparse specification by its nature.  Also, its
current --[no-]ignore-skip-worktree-entries default is totally bogus.
```

- commands for manually tweaking paths in both the index and the working tree

- `restore`
- the restore-like half of `checkout`

```
These commands should be similar to add/rm/mv in that they should
only operate on the sparse specification by default, and require a
special flag to operate on all files.

Also, note that these commands currently have a number of issues (see
the "Known bugs" section below)
```

- Commands that significantly differ for behavior A vs. behavior B:

- commands that query history

- diff (with --cached or REVISION arguments)
- grep (with --cached or REVISION arguments)
- show (when given commit arguments)
- blame (only matters when one or more -C flags are passed)
  - and annotate
- log
- whatchanged (may not exist anymore)
- ls-files
- diff-index
- diff-tree
- ls-tree

```
Note: for log and whatchanged, revision walking logic is unaffected
but displaying of patches is affected by scoping the command to the
sparse-checkout.  (The fact that revision walking is unaffected is
```

```
why rev-list, shortlog, show-branch, and bisect are not in this
list.)

ls-files may be slightly special in that e.g. `git ls-files -t` is
often used to see what is sparse and what is not.  Perhaps -t should
always work on the full tree?
```

- Commands I don’t know how to classify

- range-diff

```
Is this like `log` or `format-patch`?
```

- cherry

```
See range-diff
```

- Commands unaffected by sparse-checkouts

- shortlog
- show-branch
- rev-list
- bisect

- branch
- describe
- fetch
- gc
- init
- maintenance
- notes
- pull (merge & rebase have the necessary changes)
- push
- submodule
- tag

- config
- filter-branch (works in separate checkout without sparse-checkout setup)
- pack-refs
- prune
- remote
- repack
- replace

- bugreport

- count-objects
- fsck
- gitweb
- help
- instaweb
- merge-tree (doesn’t touch worktree or index, and merges always compute full-tree)
- rerere
- verify-commit
- verify-tag

- commit-graph
- hash-object
- index-pack
- mktag
- mktree
- multi-pack-index
- pack-objects
- prune-packed
- symbolic-ref
- unpack-objects
- update-ref
- write-tree (operates on index, possibly optimized to use sparse dir entries)

- for-each-ref
- get-tar-commit-id
- ls-remote
- merge-base (merges are computed full tree, so merge base should be too)
- name-rev
- pack-redundant
- rev-parse
- show-index
- show-ref
- unpack-file
- var
- verify-pack

- <Everything under ’Interacting with Others’ in ’git help --all’>
- <Everything under ’Low-level…Syncing’ in ’git help --all’>
- <Everything under ’Low-level…Internal Helpers’ in ’git help --all’>
- <Everything under ’External commands’ in ’git help --all’>

- Commands that might be affected, but who cares?

- merge-file
- merge-index
- gitk?

Behavior classes ==

From the above there are a few classes of behavior:

- "restrict"

```
Commands in this class only read or write files in the working tree
within the sparse specification.

When moving to a new commit (e.g. switch, reset --hard), these commands
may update index files outside the sparse specification as of the start
of the operation, but by the end of the operation those index files
will match HEAD again and thus those files will again be outside the
sparse specification.

When paths are explicitly specified, these paths are intersected with
the sparse specification and will only operate on such paths.
(e.g. `git restore [--staged] -- '*.png'`, `git reset -p -- '*.md'`)

Some of these commands may also attempt, at the end of their operation,
to cull transient differences between the sparse specification and the
sparsity patterns (see "Sparse specification vs. sparsity patterns" for
details, but this basically means either removing unmodified files not
matching the sparsity patterns and marking those files as
SKIP_WORKTREE, or vivifying files that match the sparsity patterns and
marking those files as !SKIP_WORKTREE).
```

- "restrict modulo conflicts"

```
Commands in this class generally behave like the "restrict" class,
except that:

(1) they will ignore the sparse specification and write files with
    conflicts to the working tree (thus temporarily expanding the
    sparse specification to include such files.)
(2) they are grouped with commands which move to a new commit, since
    they often create a commit and then move to it, even though we
    know there are many exceptions to moving to the new commit.  (For
    example, the user may rebase a commit that becomes empty, or have
```

```
    a cherry-pick which conflicts, or a user could run `merge
    --no-commit`, and we also view `apply --index` kind of like `am
    --no-commit`.)  As such, these commands can make changes to index
    files outside the sparse specification, though they'll mark such
    files with SKIP_WORKTREE.
```

- "restrict also specially applied to untracked files"

```
Commands in this class generally behave like the "restrict" class,
except that they have to handle untracked files differently too, often
because these commands are dealing with files changing state between
'tracked' and 'untracked'.  Often, this may mean printing an error
message if the command had nothing to do, but the arguments may have
referred to files whose tracked-ness state could have changed were it
not for the sparsity patterns excluding them.
```

- "no restrict"

```
Commands in this class ignore the sparse specification entirely.
```

- "restrict or no restrict dependent upon behavior A vs. behavior B"

```
Commands in this class behave like "no restrict" for folks in the
behavior B camp, and like "restrict" for folks in the behavior A camp.
However, when behaving like "restrict" a warning of some sort might be
provided that history queries have been limited by the sparse-checkout
specification.
```

Subcommand-dependent defaults ==

Note that we have different defaults depending on the command for the desired behavior :

- Commands defaulting to "restrict":

- diff-files
- diff (without --cached or REVISION arguments)
- grep (without --cached or REVISION arguments)

- switch
- checkout (the switch-like half)
- reset (<commit>)

- restore
- checkout (the restore-like half)
- checkout-index
- reset (with pathspec)

```
This behavior makes sense; these interact with the working tree.
```

- Commands defaulting to "restrict modulo conflicts":

- merge
- rebase
- cherry-pick
- revert

- am
- apply --index (which is kind of like an `am` `--no-commit`)

- read-tree (especially with -m or -u; is kind of like a --no-commit merge)
- reset (<tree-ish>, due to similarity to read-tree)

```
These also interact with the working tree, but require slightly
different behavior either so that (a) conflicts can be resolved or (b)
because they are kind of like a merge-without-commit operation.


(See also the "Known bugs" section below regarding `am` and `apply`)
```

- Commands defaulting to "no restrict":

- archive
- bundle
- commit
- format-patch
- fast-export
- fast-import
- commit-tree

- stash
- apply (without `--index`)

```
These have completely different defaults and perhaps deserve the most
detailed explanation:
```

```
In the case of commands in the first group (format-patch,
fast-export, bundle, archive, etc.), these are commands for
communicating history, which will be broken if they restrict to a
subset of the repository.  As such, they operate on full paths and
have no `--restrict` option for overriding.  Some of these commands may
take paths for manually restricting what is exported, but it needs to
be very explicit.

In the case of stash, it needs to vivify files to avoid losing the
user's changes.

In the case of apply without `--index`, that command needs to update
the working tree without the index (or the index without the working
tree if `--cached` is passed), and if we restrict those updates to the
sparse specification then we'll lose changes from the user.
```

- Commands defaulting to "restrict also specially applied to untracked files":

- add
- rm
- mv
- update-index
- status
- clean (?)

```
        Our original implementation for the first three of these commands was
        "no restrict", but it had some severe usability issues:

          * `git add <somefile>` if honored and outside the sparse
        specification, can result in the file randomly disappearing later
        when some subsequent command is run (since various commands
        automatically clean up unmodified files outside the sparse
        specification).
          * `git rm '*.jpg'` could very negatively surprise users if it deletes
        files outside the range of the user's interest.
          * `git mv` has similar surprises when moving into or out of the cone,
        so best to restrict by default

        So, we switched `add` and `rm` to default to "restrict", which made
        usability problems much less severe and less frequent, but we still got
```

```
        complaints because commands like:

        git add <file-outside-sparse-specification>
        git rm <file-outside-sparse-specification>

        would silently do nothing.  We should instead print an error in those
        cases to get usability right.

        update-index needs to be updated to match, and status and maybe clean
        also need to be updated to specially handle untracked paths.

        There may be a difference in here between behavior A and behavior B in
        terms of verboseness of errors or additional warnings.
```

- Commands falling under "restrict or no restrict dependent upon behavior A vs. behavior B"

- diff (with --cached or REVISION arguments)
- grep (with --cached or REVISION arguments)
- show (when given commit arguments)
- blame (only matters when one or more -C flags passed)
  - and annotate
- log
  - and variants: shortlog, gitk, show-branch, whatchanged, rev-list
- ls-files
- diff-index
- diff-tree
- ls-tree

```
For now, we default to behavior B for these, which want a default of
"no restrict".

Note that two of these commands -- diff and grep -- also appeared in a
different list with a default of "restrict", but only when limited to
searching the working tree.  The working tree vs. history distinction
is fundamental in how behavior B operates, so this is expected.  Note,
though, that for diff and grep with --cached, when doing "restrict"
behavior, the difference between sparse specification and sparsity
patterns is important to handle.

"restrict" may make more sense as the long term default for these[12].
Also, supporting "restrict" for these commands might be a fair amount
```

```
of work to implement, meaning it might be implemented over multiple
releases.  If that behavior were the default in the commands that
supported it, that would force behavior B users to need to learn to
slowly add additional flags to their commands, depending on git
version, to get the behavior they want.  That gradual switchover would
be painful, so we should avoid it at least until it's fully
implemented.
```

Sparse specification vs. sparsity patterns ==

In a well-behaved situation, the sparse specification is given directly by the $GIT_DIR/info/sparse-checkout file. However, it can transiently diverge for a few reasons:

- needing to resolve conflicts (merging will vivify conflicted files)
- running Git commands that implicitly vivify files (e.g. "git stash apply")
- running Git commands that explicitly vivify files (e.g. "git checkout --ignore-skip-worktree-bits FILENAME")
- other commands that write to these files (perhaps a user copies it from elsewhere)

For the last item, note that we do automatically clear the SKIP_WORKTREE bit for files that are present in the working tree. This has been true since 82386b4496 ("Merge branch ’en/present-despite-skipped’", 2022-03-09)

However, such a situation is transient because:

- Such transient differences can and will be automatically removed as a side-effect of commands which call unpack_trees() (checkout, merge, reset, etc.).
- Users can also request such transient differences be corrected via running `git sparse-checkout reapply`. Various places recommend running that command.
- Additional commands are also welcome to implicitly fix these differences; we may add more in the future.

While we avoid dropping unstaged changes or files which have conflicts, we otherwise aggressively try to fix these transient differences. If users want these differences to persist, they should run the `set` or `add` subcommands of `git sparse-checkout` to reflect their intended sparse specification.

However, when we need to do a query on history restricted to the "relevant subset of files" such a transiently expanded sparse specification is ignored. There are a couple reasons for this:

- The behavior wanted when doing something like git grep expression REVISION is roughly what the users would expect from git checkout REVISION && git grep expression (modulo a "REVISION:" prefix), which has a couple ramifications:

- REVISION may have paths not in the current index, so there is no path we can consult for a SKIP_WORKTREE setting for those paths.

- Since `checkout` is one of those commands that tries to remove transient differences in the sparse specification, it makes sense to use the corrected sparse specification (i.e. $GIT_DIR/info/sparse-checkout) rather than attempting to consult SKIP_WORKTREE anyway.

So, a transiently expanded (or restricted) sparse specification applies to the working tree, but not to history queries where we always use the sparsity patterns. (See [16] for an early discussion of this.)

Similar to a transiently expanded sparse specification of the working tree based on additional files being present in the working tree, we also need to consider additional files being modified in the index. In particular, if the user has staged changes to files (relative to HEAD) that do not match the sparsity patterns, and the file is not present in the working tree, we still want to consider the file part of the sparse specification if we are specifically performing a query related to the index (e.g. git diff --cached [REVISION], git diff-index [REVISION], git restore --staged --source=REVISION — PATHS, etc.) Note that a transiently expanded sparse specification for the index usually only matters under behavior A, since under behavior B index operations are lumped with history and tend to operate full-tree.

Implementation Questions

==

- Do the options --scope={sparse,all} sound good to others? Are there better options?

- Names in use, or appearing in patches, or previously suggested:

- --sparse/--dense
- --ignore-skip-worktree-bits
- --ignore-skip-worktree-entries
- --ignore-sparsity
- --[no-]restrict-to-sparse-paths
- --full-tree/--sparse-tree
- --[no-]restrict
- --scope={sparse,all}

- --focus/--unfocus
- --limit/--unlimited

- Rationale making me lean slightly towards --scope={sparse,all}:

- We want a name that works for many commands, so we need a name that does not conflict
- We know that we have more than two possible usecases, so it is best to avoid a flag that appears to be binary.
- --scope={sparse,all} isn't overly long and seems relatively explanatory
- `--sparse`, as used in add/rm/mv, is totally backwards for grep/log/etc. Changing the meaning of `--sparse` for these commands would fix the backwardness, but possibly break existing scripts. Using a new name pairing would allow us to treat `--sparse` in these commands as a deprecated alias.
- There is a different `--sparse`/`--dense` pair for commands using revision machinery, so using that naming might cause confusion
- There is also a `--sparse` in both pack-objects and show-branch, which don't conflict but do suggest that `--sparse` is overloaded
- The name --ignore-skip-worktree-bits is a double negative, is quite a mouthful, refers to an implementation detail that many users may not be familiar with, and we'd need a negation for it which would probably be even more ridiculously long. (But we can make --ignore-skip-worktree-bits a deprecated alias for --no-restrict.)

- If a config option is added (sparse.scope?) what should the values and description be? "sparse" (behavior A), "worktree-sparse-history-dense" (behavior B), "dense" (behavior C)? There's a risk of confusion, because even for Behaviors A and B we want some commands to be full-tree and others to operate sparsely, so the wording may need to be more tied to the usecases and somehow explain that. Also, right now, the primary difference we are focusing is just the history-querying commands (log/diff/grep). Previous config suggestion here: [13]

- Is `--no-expand` a good alias for ls-files's `--sparse` option? (`--sparse` does not map to either `--scope=sparse` or `--scope=all`, because in non-cone mode it does nothing and in cone-mode it shows the sparse directory entries which are technically outside the sparse specification)

- Under Behavior A:

- Does ls-files' `--no-expand` override the default `--scope=all`, or does it need an extra flag?
- Does ls-files' `-t` option imply `--scope=all`?
- Does update-index's `--[no-]skip-worktree` option imply `--scope=all`?

- sparse-checkout: once behavior A is fully implemented, should we take an interim measure to ease people into switching the default? Namely, if folks are not already in a sparse checkout, then require `sparse-checkout` `init/set` to take a `--set-scope=(sparse|worktree-sparse-history-dense|dense)` flag (which would set sparse.scope

according to the setting given), and throw an error if the flag is not provided? That error would be a great place to warn folks that the default may change in the future, and get them used to specifying what they want so that the eventual default switch is seamless for them.

Implementation Goals/Plans ==

- Get buy-in on this document in general.
- Figure out answers to the 'Implementation Questions' sections (above)
- Fix bugs in the 'Known bugs' section (below)
- Provide some kind of method for backfilling the blobs within the sparse specification in a partial clone

```
[Below here is kind of spitballing since the first two haven't been resolved]
```

- update-index: flip the default to --no-ignore-skip-worktree-entries, nuke this stupid "Oh, there's a bug? Let me add a flag to let users request that they not trigger this bug." flag
- Flags & Config
- Make `--sparse` in add/rm/mv a deprecated alias for `--scope=all`
- Make `--ignore-skip-worktree-bits` in checkout-index/checkout/restore a deprecated aliases for `--scope=all`
- Create config option (sparse.scope?), tie it to the "Cliff notes" overview
- Add --scope=sparse (and --scope=all) flag to each of the history querying commands. IMPORTANT: make sure diff machinery changes don't mess with format-patch, fast-export, etc.

Known bugs ==

This list used to be a lot longer (see e.g. [1,2,3,4,5,6,7,8,9]), but we've been working on it.

1. Behavior A is not well supported in Git. (Behavior B didn't used to be either, but was the easier of the two to implement.)

2. am and apply:

```
apply, without `--index` or `--cached`, relies on files being present
in the working copy, and also writes to them unconditionally.  As
such, it should first check for the files' presence, and if found to
be SKIP_WORKTREE, then clear the bit and vivify the paths, then do
its work.  Currently, it just throws an error.

apply, with either `--cached` or `--index`, will not preserve the
SKIP_WORKTREE bit.  This is fine if the file has conflicts, but
otherwise SKIP_WORKTREE bits should be preserved for --cached and
probably also for --index.

am, if there are no conflicts, will vivify files and fail to preserve
the SKIP_WORKTREE bit.  If there are conflicts and `-3` is not
specified, it will vivify files and then complain the patch doesn't
apply.  If there are conflicts and `-3` is specified, it will vivify
files and then complain that those vivified files would be
overwritten by merge.
```

3. reset --hard:

```
reset --hard provides confusing error message (works correctly, but
misleads the user into believing it didn't):

$ touch addme
$ git add addme
$ git ls-files -t
H addme
H tracked
S tracked-but-maybe-skipped
$ git reset --hard                          # usually works great
error: Path 'addme' not uptodate; will not remove from working tree.
HEAD is now at bdbbb6f third
$ git ls-files -t
H tracked
S tracked-but-maybe-skipped
$ ls -1
tracked
```

```
`git reset --hard` DID remove addme from the index and the working tree, contrary
to the error message, but in line with how reset --hard should behave.
```

4. read-tree

```
`read-tree` doesn't apply the 'SKIP_WORKTREE' bit to *any* of the
entries it reads into the index, resulting in all your files suddenly
appearing to be "deleted".
```

5. Checkout, restore:

```
These command do not handle path & revision arguments appropriately:

$ ls
tracked
$ git ls-files -t
H tracked
S tracked-but-maybe-skipped
$ git status --porcelain
$ git checkout -- '*skipped'
error: pathspec '*skipped' did not match any file(s) known to git
$ git ls-files -- '*skipped'
tracked-but-maybe-skipped
$ git checkout HEAD -- '*skipped'
error: pathspec '*skipped' did not match any file(s) known to git
$ git ls-tree HEAD | grep skipped
100644 blob 276f5a64354b791b13840f02047738c77ad0584f    tracked-but-maybe-skipped
$ git status --porcelain
$ git checkout HEAD~1 -- '*skipped'
$ git ls-files -t
H tracked
H tracked-but-maybe-skipped
$ git status --porcelain
M  tracked-but-maybe-skipped
$ git checkout HEAD -- '*skipped'
$ git status --porcelain
$

Note that checkout without a revision (or restore --staged) fails to
find a file to restore from the index, even though ls-files shows
```

```
such a file certainly exists.

Similar issues occur with HEAD (--source=HEAD in restore's case),
but suddenly works when HEAD~1 is specified.  And then after that it
will work with HEAD specified, even though it didn't before.

Directories are also an issue:

$ git sparse-checkout set nomatches
$ git status
On branch main
You are in a sparse checkout with 0% of tracked files present.

nothing to commit, working tree clean
$ git checkout .
error: pathspec '.' did not match any file(s) known to git
$ git checkout HEAD~1 .
Updated 1 path from 58916d9
$ git ls-files -t
S tracked
H tracked-but-maybe-skipped
```

6. checkout and restore --staged, continued:

```
These commands do not correctly scope operations to the sparse
specification, and make it worse by not setting important SKIP_WORKTREE
bits:

$ git restore --source OLDREV --staged outside-sparse-cone/
$ git status --porcelain
MD outside-sparse-cone/file1
MD outside-sparse-cone/file2
MD outside-sparse-cone/file3

We can add a --scope=all mode to `git restore` to let it operate outside
the sparse specification, but then it will be important to set the
SKIP_WORKTREE bits appropriately.
```

7. Performance issues; see:

```
https://lore.kernel.org/git/CABPp-
BEkJQoKZsQGCYioyga_uoDQ6iBeW+FKr8JhyuuTMK1RDw@mail.gmail.com/
```

Reference Emails ==

Emails that detail various bugs we've had in sparse-checkout:

[1] (Original descriptions of behavior A & behavior B):

https://lore.kernel.org/git/CABPp-BGJ_Nvi5TmgriD9Bh6eNXE2EDq2f8e8QKXAeYG3BxZafA@mail.gmail.com/

[2] (Fix stash applications in sparse checkouts; bugs from behavioral differences):

https://lore.kernel.org/git/ccfedc7140dbf63ba26a15f93bd3885180b26517.1606861519.git.gitgitgadget@gmail.com/

[3] (Present-despite-skipped entries):

https://lore.kernel.org/git/11d46a399d26c913787b704d2b7169cafc28d639.1642175983.git.gitgitgadget@gmail.com/

[4] (Clone --no-checkout interaction):

https://lore.kernel.org/git/pull.801.v2.git.git.1591324899170.gitgitgadget@gmail.com/ (clone --no-checkout)

[5] (The need for update_sparsity() and avoiding `read-tree` `-mu` `HEAD`):

https://lore.kernel.org/git/3a1f084641eb47515b5a41ed4409a36128913309.1585270142.git.gitgitgadget@gmail.com/

[6] (SKIP_WORKTREE is advisory, not mandatory):

https://lore.kernel.org/git/844306c3e86ef67591cc086decb2b760e7d710a3.1585270142.git.gitgitgadget@gmail.com/

[7] (`worktree` `add` should copy sparsity settings from current worktree):

https://lore.kernel.org/git/c51cb3714e7b1d2f8c9370fe87eca9984ff4859f.1644269584.git.gitgitgadget@gmail.com/

[8] (Avoid negative surprises in add, rm, and mv):

- https://lore.kernel.org/git/cover.1617914011.git.matheus.bernardino@usp.br/
- https://lore.kernel.org/git/pull.1018.v4.git.1632497954.gitgitgadget@gmail.com/

[9] (Move from out-of-cone to in-cone):

- https://lore.kernel.org/git/20220630023737.473690-6-shaoxuan.yuan02@gmail.com/
- https://lore.kernel.org/git/20220630023737.473690-4-shaoxuan.yuan02@gmail.com/

[10] (Unnecessarily downloading objects outside sparse specification):

https://lore.kernel.org/git/CAOLTT8QfwOi9yx_qZZgyGa8iL8kHWutEED7ok_jxwTcYT_hf9Q@mail.gmail.com/

[11] (Stolee's comments on high-level usecases):

https://lore.kernel.org/git/1a1e33f6-3514-9afc-0a28-5a6b85bd8014@gmail.com/

[12] Others commenting on eventually switching default to behavior A:

- https://lore.kernel.org/git/xmqqh719pcoo.fsf@gitster.g/
- https://lore.kernel.org/git/xmqqzgeqw0sy.fsf@gitster.g/
- https://lore.kernel.org/git/a86af661-cf58-a4e5-0214-a67d3a794d7e@github.com/

[13] Previous config name suggestion and description:

```
https://lore.kernel.org/git/CABPp-
BE6zW0nJSStcVU=_DoDBnPgLqOR8pkTXK3dW11=T01OhA@mail.gmail.com/
```

[14] Tangential issue: switch to cone mode as default sparse specification mechanism:

https://lore.kernel.org/git/a1b68fd6126eb341ef3637bb93fedad4309b36d0.1650594746.git.gitgitgadget@gmail.com/

[15] Lengthy email on grep behavior, covering what should be searched:

https://lore.kernel.org/git/CABPp-BGVO3QdbfE84uF_3QDF0-y2iHHh6G5FAFzNRfeRitkuHw@mail.gmail.com/

[16] Email explaining sparsity patterns vs. SKIP_WORKTREE and history operations, search for the parenthetical comment starting "We do not check".

https://lore.kernel.org/git/CABPp-BFsCPPNOZ92JQRJeGyNd0e-TCW-LcLyr0i_+VSQJP+GCg@mail.gmail.com/

[17] https://lore.kernel.org/git/20220207190320.2960362-1-jonathantanmy@google.com/

□□□ #2
□ □□□ □□□ 2026-10-31 20:17:18 UTC
□ □□□ □□□ 2026-10-31 20:17:21 UTC