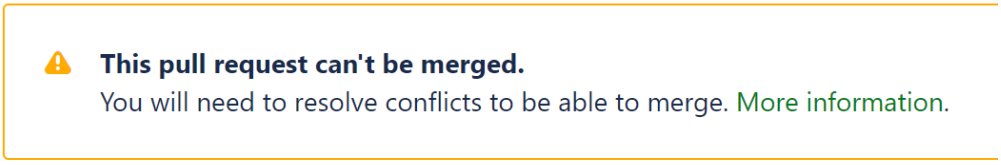
How to resolve a PR merge conflict

You just submitted a PR on Stash that fixed the fertility model from multiplying the number of live births in each country by an extra 1000. Hype. But...uh oh...what is this?



This is not good. How are you going to get these super important changes to the fertility model through to the rest of the team now?

In this document, we'll walk through why this is happening, how to understand and address the problem, and adjustments to coding practices you can make to avoid these types of situations.

What is going on?

What we have here is a merge conflict. Atlassian has a really great explanation for what a merge conflict is here. Basically, the PR you are attempting to submit made changes to a section of a file that had also been changed on the remote master branch (where you're trying to merge your changes into). Git doesn't know how to address simultaneous changes to the same file, and it has asked you to take responsibility for the conflict.

How could this have happened? There are a couple of likely scenarios

- When you first made your local feature branch off of your local master (Step 2), your local master wasn't up to date with the remote master. You might have then made changes to a file that was changed in the remote master branch, that wasn't being tracked on your local master. It's important to run

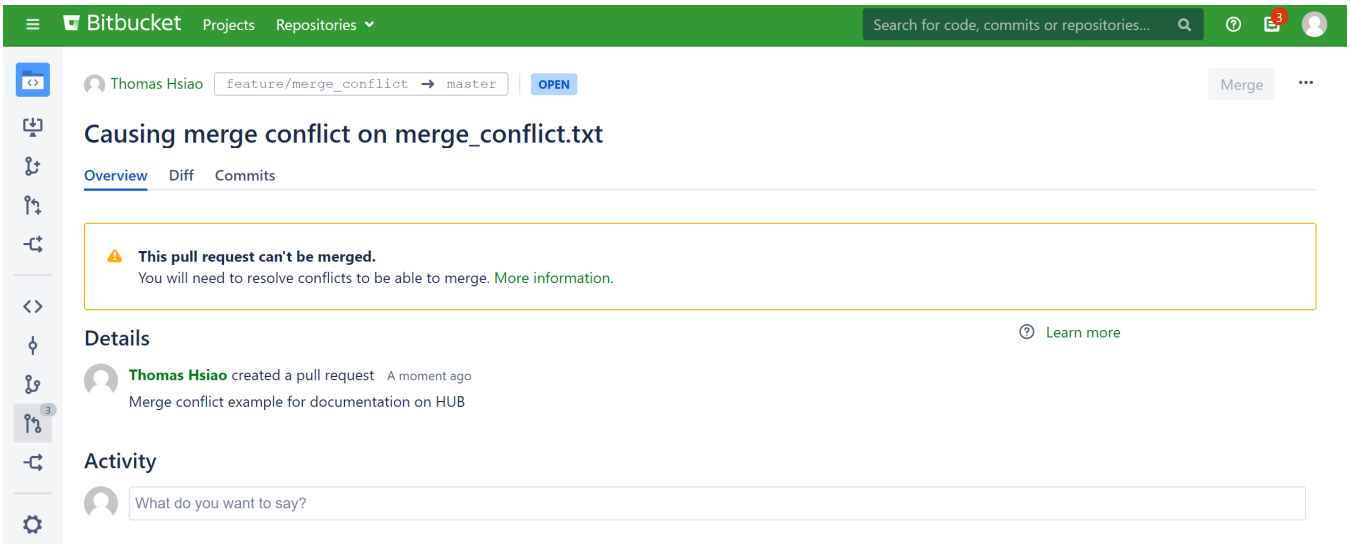
  ```
  $ git checkout master
  $ git pull origin master
  ```

  before you start any new feature branch to ensure you're working off of the latest and greatest codebase.
- You might have synced your local master with the latest and greatest remote master before you made your new feature branch, but while you were working on your PR, Charlton off in his own time successfully merged his PR into remote master. If Charlton modified a file that you are currently working on, then eventually your new PR will conflict with the remote master, which now has Charlton's PR incorporated. Communicating with team members about what files will be changed before beginning on features can help a lot with avoiding these sorts of merge conflicts.

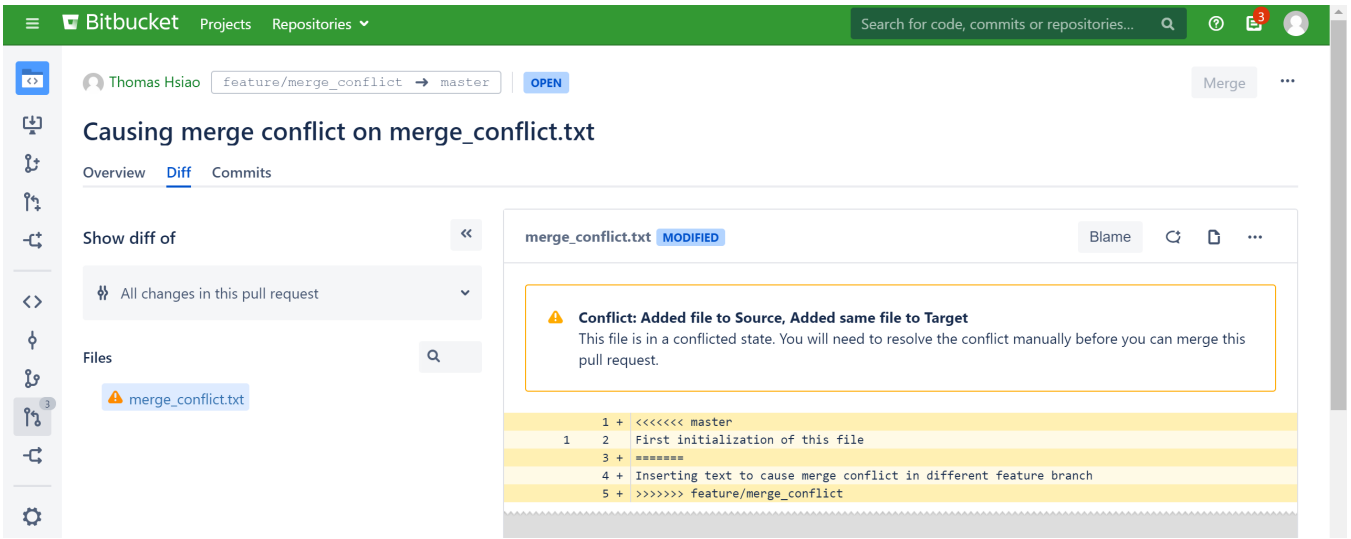
Let's walk through a small example to get an idea of the steps to solve this problem.

Understanding the Problem

Let's take a look at the PR page on Stash. If you are currently trying to resolve your own merge conflict, it will probably look similar to this with your own title and description of your PR. Stash helpfully tells us we can't merge our pull request, but we need more information.



If we click on the "Diff" Tab next to "Overview", we can see specifically which files and which regions of those files are experiencing the merge conflict.



In this case, only one file, **merge_conflict.txt** has conflicting commits. On the right we can see the actual contents of the file. Only the text highlighted in yellow is the source of the merge conflict (in this example, all of the text is involved in the merge conflict but that is not always the case). We see the block of text is divided by three strange lines, "<<<<<<<<< master", "=========", and ">>>>>>>>>> feature/merge_conflict". The row of equal signs is a divider that separates the changes made to a region of the file in one branch vs. another. For this example, the master branch only has "First initialization of this file" as text in merge_conflict.txt, while the feature/merge_conflict branch has "Inserting text to cause merge conflict in different feature branch" as the only text in merge_conflict.txt. A direct conflict! If the feature/merge_conflict were to have had both lines in the text file, with "First initialization of this file" first (as it is in the master branch), there would be no conflict. Git is very smart in identifying what these specific regions are, and now all we have to do is edit the text to tell Git what our actual commit should look like.

Solution

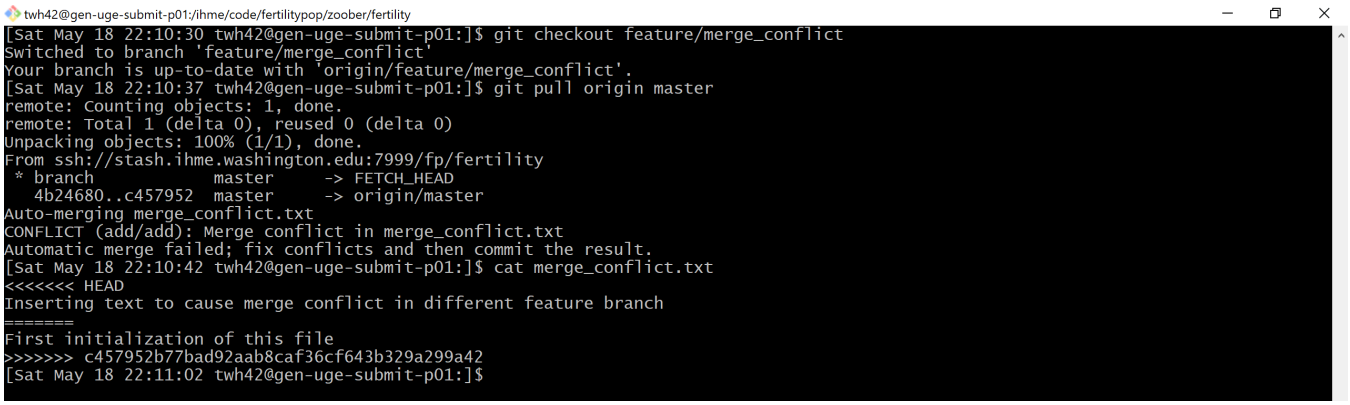
To solve merge conflicts, we want to do all editing in our local repository. There is an option to resolve these in the remote on Stash, but it's cleaner to have the owner of the PR resolve things on their local repo and just update their existing PR with the fixes. However, right now only the remote on Stash knows that there is a merge conflict. Our local isn't aware of any merge conflicts yet. To be able to see where the merge conflicts are on our local repo we can run the following code

```
$ git checkout <feature branch name>
$ git pull origin master


<resolve merge conflicts and do usual stage/commit>


$ git push origin <feature branch name>
```

What does this do?

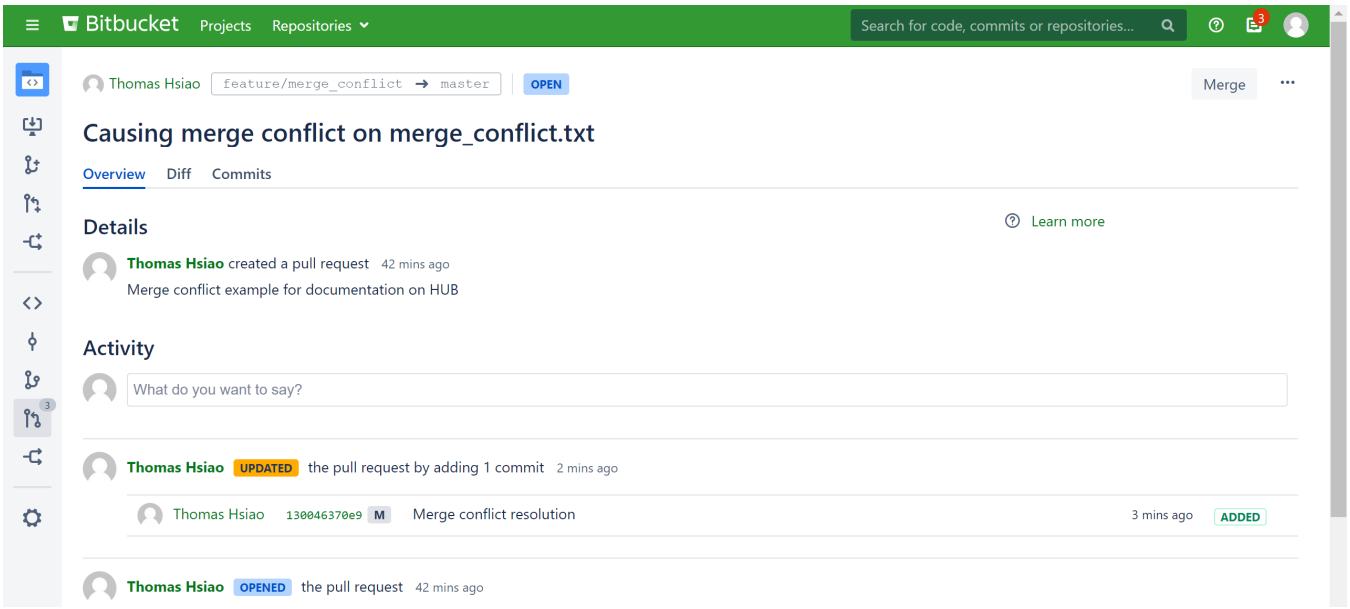
1. Switch to the local feature branch
2. Merge the remote master branch into our local feature branch. This will replicate what you saw on the "Diff" tab of the Stash page in your local repo.
3. Manually resolve all the merge conflicts
4. Push the resolutions to the remote feature branch, thereby updating the PR.

Here is an example of what that should look like from the terminal, up to Step 2. When we look at the contents of merge_conflict.txt using the **cat** bash command from the local feature branch, we that it has the strange divider lines and branch labels that we saw on Stash! Success!

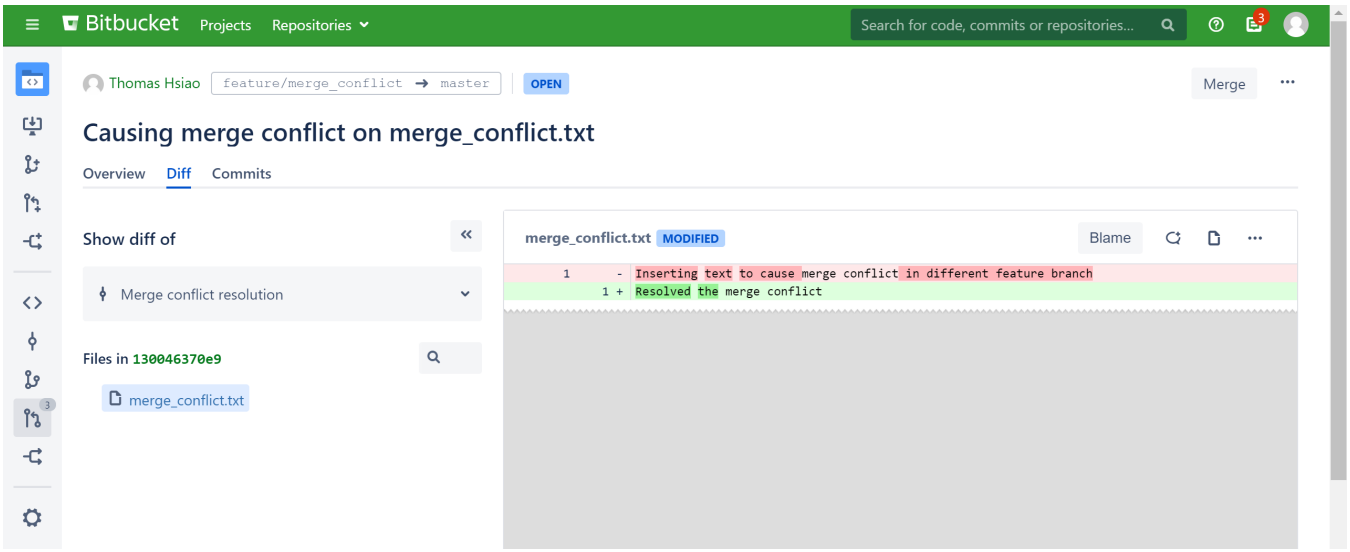


We'll edit the file now by deleting everything currently in there and replacing it with "Resolved merge conflict". Keep in mind, however you resolve your merge conflict Git will accept. You could keep changes only from one branch or the other, delete changes from both, or even keep both of the changes. It's totally up to you to figure out what needs to be done.

Once we've edited our file, staged and committed it, and pushed our changes up to the remote feature branch again, we can take look at our updated PR on Stash now.



The scary yellow warning message telling us we can't merge our PR is now gone! We see that a new commit has been added to the PR, with the message I attached to it "Merge conflict resolution". Seems like our fix went through, but let's check for sure under the "Diff" tab.



All the yellow highlighted text is now gone, replaced with red and green. Green means additions to the file, and red means removal. This reflects the changes we made to the file to resolve the merge conflict, namely deleting all of the text and replacing it with "Resolved the merge conflict". Now that our merge conflict is out of the way, we can proceed regularly with the standard PR review process!

Once you finish up the PR process, you'll want to remember to sync up your local master with the remote master since that is likely the reason why you started off with a merge conflict in the first place. Just run

```
$ git checkout master
$ git pull origin master
```

ASIDE: You may be wondering, why isn't the text originally in the master branch to this file, "First initialization of this file" not being shown in red as being removed? Remember, this is a PR for the remote feature branch, so only text from the remote feature branch will be shown to be updated. Only the local feature branch pulled in that text from the remote master because we explicitly called **git pull origin master** while on the local feature branch in our working directory.

Lessons

Based on the example, here are some best practices to avoid unnecessary merge conflicts

1. Regularly sync the local master with the remote master before creating new feature branches.
2. Communicate with team members to reduce the number of times you simultaneously work on the same regions of the same files
3. Make your PR's small and succinct, not gigantic behemoths of scripts. Can you imagine having to manually resolving all of those merge conflicts if they were to span 20 files and hundreds of lines of code?