Review of Patch Reviewing

Stephen Frost
Crunchy Data
stephen@crunchydata.com

PGConf.EU 2018
October 24, 2018

Stephen Frost

- Chief Technology Officer @ Crunchy Data
- Committer, PostgreSQL
- Major Contributor, PostgreSQL
- PostgreSQL Infrastructure Team
- Default roles
- Row-Level Security in 9.5
- Column-level privileges in 8.4
- Implemented the roles system in 8.3
- Contributions to PL/pgSQL, PostGIS

Reviewing Patches

- Slightly different set of skills from C programming
- Really helps to know C, but not required
- Break code, not write code (mostly)
- Patch submitters
  - Expect and plan to do reviews!
  - Committers make year-long schedules to review/apply patches
- PostgreSQL has 'CommitFests'
- An attempt to manage the volume

But Committing is Easy!

- Why is committing a patch hard?
  - Not really the commit
  - Each patch needs to be carefully reviewed
  - Often refactoring should be done
  - Sometimes parts need to be rewritten (comments...)
- Commitfests try to push it to the author
  - Code base improves when commits start out excellent
  - Authors should be learning from feedback
  - More work on patch authors reduces time committers spend
  - Reviewing good code is faster

The Bar (no, not that bar)

- PostgreSQL code is very high quality
  - Need to keep it that way!
  - Lots of comments help
- The onus/expectation is the committer gets code which meets this level
- Reviewers are part of the system to make this happen
- Not the reviewer's job to fix the patch!
- Different patches require different amounts of work
- How far from the bar is it..?

What are Commitfests?

- A month (typically) set aside by committers to review other people's code
  - Patches submitted by anyone
  - Other committers' code too
- PostgreSQL 12 Commitfests
  - July 2018 - 200 Patches
  - September 2018 - 215 Patches
  - November 2018 - Already 179!
  - January 2019
  - March 2019
- Non-committers review patches first
- Patch review is iterative during the CommitFest
- Only when reviewer is happy does it go to committer

Tracking it all

`https://commitfest.postgresql.org`

- We have a website for that!
- Community accounts
  - Need a community account to make changes
  - Useful for lots of other PG sites too!
  - `https://www.postgresql.org/account/`
- Based on / integrated with mailing lists
  - Authors post patches to -hackers
  - Authors then add patch to commitfest app with link to -hackers
  - Reviews posted to the patch thread
  - Commits go to -committers

Commitfest Statuses

- Needs Review (pick it up!)
  - You can add yourself!
  - Click on the patch, then 'Become Reviewer'
  - Try to only claim a patch to review if you are actively working on it
- Waiting on Author
  - Indicates patch has been reviewed
  - Patch review asked author to make changes
  - Authors should try to minimize time in this state
- Ready for Committer
  - Patch reviewer feels patch is ready for commit
  - Should not be set by patch author generally
- Others: Committed, Returned w/ Feedback, Rejected

Steps to reviewing a patch

`https://wiki.postgresql.org/wiki/Reviewing_a_Patch`

- Submission review (correct format, et al)
- Usability/Feature Functionality review
- Performance review
- Code Review
- Architecture review
- Lastly, post the review and update status

Submission Review

- Patch sent to -hackers?
- Correct format (context diff), or git format-patch
- No pull requests!
- Review for completeness
  - Description of the patch/change
  - Updates documentation, if appropriate
  - Regression tests!
  - pg_dump/pg_restore support

Style Review

- Check the PostgreSQL Style guides
  - Documentation style guide
  - `https://www.postgresql.org/docs/current/static/docguide-style.html`
  - Error message style guide
  - `https://www.postgresql.org/docs/current/static/error-style-guide.html`
- Review documentation and error messages to make sure they comply

Documentation Style

- Each command has a reference page for it
- Should include
  - Name (of the command...)
  - Synopsis
  - Description
  - Options, as appropriate
  - Exit Status, if a command line tool
  - Usage
  - Environment variables, if appropriate
  - Files used, if appropriate
  - Notes / Examples / History / See Also
  - Author, but only in the contrib section

Error Message Style

- There are three main parts to an error message
  - Primary message
  - Detail information
  - Hint, if appropriate
- Check that these are included
- Also look for SQL state, if appropriate
- Should not make assumptions about formatting
- Should not end an error message with a newline
- Double-quotes should be used when quoting
- Quotes used for filenames, user identifiers, and other variables
- Should not use passive voice- use active voice, PostgreSQL is not a human

Usability and Feature Functionality Review

- Does it do what it says it does?
  - Read updated documentation
  - Review Regression tests!
  - Check that they all pass (make check)
  - See if they make sense!
  - Consider if more should be added
- Any compiler warnings? (There should not be)
- Try to make PG crash. ;)

Performance Review

- Stress test the patch for weeks!
  - Ok, no, maybe not
  - Review Regression tests!
- Does the patch claim to improve performance
  - Look for / ask for performance tests
  - Ask about the worst case
  - Consider if this makes some cases slower
  - If possible, confirm/retest benchmarks
- If patch is a new feature but very slow, ask the author

Code Review - Format

- Tab-based alignment (4-column hard tabs)
- No cuddling braces (brackets- {, }, get their own lines)
- 80-character lines (use pg_indent!)
- No C++ style comments (no // comment)
- C99, with some caveats
  - Ok, no, maybe not
  - Review Regression tests!
- Consider code flow, style
- Should look like one author wrote the section

Code Review - Look at the code!

- Existing subsystems should be used
  - We don't need/want 5 different linked list implementations
  - (we already have 3)
  - Be familiar with what we have: backend/{libs,utils}/ ; common/
  - No direct malloc() calls in backend, should use palloc()!
  - Use existing macros!
- Comments, comments, comments
  - Look for good comments, look for 'XXX'/'TODO'/'NOTE's also
  - Functions should have a multi-line comment block above
  - Did function args change (or what they do..)? comment better change!
  - Comments should precede every major if/while/etc block
  - Should answer the 'Why are we doing this?'
  - We can read what the code is doing- no need to say it again

Code Review - functions and more...

- Use functions if possible!
  - Could a big block be functionalized?
  - Clear inputs/outputs for the block
  - Maybe parts could be broken out as functions?
  - Is the code common enough to be useful elsewhere?
- Watch for duplicated code- probably should be functions
- Look for copy/paste happening
  - Check for patterns in differences
  - Consider if any places might have been missed
  - New structure member added? Check over structure usages
  - Watch for abstraction layer violations

Architecture Review - really more code...

- PostgreSQL has defined subsystems
- APIs exist between the systems
- Look for abstractions / simplifications
- Should only include headers needed
- Committers review this carefully, so don't stress

Post Your Review!

- Post should be sent to -hackers AND CC Author, at least
- Generally we 'reply all' on PG mailing lists
- Try to maintain threading, reply to patch post
- Update commitfest application with new state
- Be specific about what you reviewed
- When suggesting changes to the patch-
  - Be specific, include actionable changes
  - Include details, such as what you did to cause a crash
- Should it be marked ready for committer?
  - Needs to be more than just "looks good to me"
  - Consider it a 'book report' of the patch
  - Review is for the author and the committer

Updating the Status

Transistions by reviewers

- Needs Review TO Waiting on author
  - Your review includes suggested changes
  - Author updates patch, then sets back to Needs Review
- Needs Review TO Returned with feedback
  - Patch not ready for commit
  - Final review done for this Commitfest
- Needs Review TO Rejected
  - Will not be accepted
  - May mean the wrong approach is used
  - Might be a "feature" the community does not want to have
- Needs Review TO Ready for Committer
  - Only once patch has been fully reviewed

Updating the Status

Transistions by authors

- Waiting on author TO Needs Review
  - Patch has been updated
  - New version needs to be reviewed again
- Ready for Committer TO Committed
  - Might be done by committer
  - Could really be done by anyone

Updating the Status

Transistions by committers

- Ready for committer TO Waiting on author
  - Committer asks for more changes
  - Review what the committer said!
  - Updated patch goes back to Needs Review
- Ready for Committer TO Committed
  - Could really be done by anyone
- Ready for Committer TO Rejected
  - Committer decides the patch is not what we want
- Ready for Committer TO Returned with feedback
  - Committer decides the patch is not ready to be committed

When the committer changes things or disagrees

- Learn from the comments made by the committer
- If committer changed the code- review what was changed
- Paticipate in the patch thread
- Do not think of review as fire and forget
- Consider if the wiki needs updating!
- Ask me to update my talk ;)

Questions?

Thanks!