A journey from Postgres enthusiast to committer

- Amit Kapila | PGCon 2019

Contents

- How I started with Postgres community
- What motivates me to keep working on it
- Start contribution
- Writing patches
- Important coding guidelines
- Feedback process
- Case study

How I started with Postgres community

- In my previous job, we were building an in-house database based on Postgres-8.3.
- I was responsible for doing enhancements in SQL and improving performance.
- While doing some research, I came across some ideas which were being discussed on pgsql-hackers and pgsql-performance.
- I subscribed to those mailing lists and starting discussing some ideas related to my work.
- As far as I can trace, my first email was related to cheaper snapshots back in 2011.

How I started with Postgres community

- In the beginning, the main intention to get involved with the community is to just learn more about Postgres which can help me in my day job.
- Based on some of my work, I proposed a paper in PGCon 2012 which got accepted.
- Here, I met with many Postgres hackers and senior members with whom I have discussed a few ideas.
- That experience has further increased my interest in working with Postgres community.
- I have started working on a few performance projects and started doing a review of other peoples patches.

How I started with Postgres community

- Slowly working with the community has improved my awareness about coding guidelines and designs of some intricate parts of Postgres.
- Then, I started getting myself involved in some design discussions for other patches on hackers.
- I do read pgsql-committers as well to be aware of all the work going on, though it is a bit difficult to read each and every commit.

What motivates me to keep working on it

- What glues me most to the community is the learning by involving myself in various kinds of work.
- Every discussion made me study and think more about the code.
- Even reading carefully other email threads on hackers helps me to learn new things.
- Over the years, I have learned a lot about bench-marking by either doing performance tests for other people's patches or doing the same for my own patches.
- Meeting other hackers and discussing the various ideas in conferences is always a learning experience.

What motivates me to keep working on it

The continuous learning and improving my patches and reviews got me commit bit last year (2018).

What motivates me to keep working on it

- I think it is somewhat similar for everyone. Basically, one has to spend a lot of time reading and modifying Postgres code.
- Even though post-commit, the main responsibility for any bugs in the commit is of a committer, but authors also get involved which is a good sign. See some examples on slide-35.
- Reviewing other peoples patches help in that especially if one can think of doing the patch in a better way.
- Reviewing and writing patches even if they finally don't get accepted will help in building committer-grade skills.
- Some of these things are mentioned by Tom Lane on the mailing list.

Start Contribution

- To start with one should be aware of community resources.
- Frequently asked questions by developers.
- Mailing lists.
- pgsql-hackers is must for source code developers.
- Blogs
- IRC channel

Start Contribution – General tips

- Ask questions on relevant lists.
- Answering questions of others with the best of your knowledge.
- Writing blogs about Postgres.
- Giving presentations in conferences.
- You can help with translations and packaging.

Start Contribution – For Developers

- What should be my first patch or first few patches?
- There are various ways to start contributing in terms of code/docs/tests
  - Start reading emails on pgsql-hackers and pgsql-bugs. Many times other hackers need help in completing their patches.
  - Try to fix the bugs reported by others on pgsql-bugs.
  - Testing the latest features in the release and report bugs if any or if you can write a test which improves coverage, that is also a good contribution.
  - Read about the features and see if you can add something which is missing about a feature and submit it as a doc patch.

Start Contribution – For Developers contd..

- Ways to start contributing
  - During commitfests, review other people patches.
  - There are times when some part of the feature/patch is decided to be done as a separate patch.
  - Todo list, unfortunately, this is not something you can directly start work upon, but it can give you some pointers about the areas in which you can start looking into.

Writing Patches

- Download the code and start understanding the patch creation process. See working with git.
- If the patch is simple like typo fix, small documentation change or some obvious code bug, then directly create the patch and send it to pgsql-hackers.
- For a feature or a non-trivial patch, before creating the actual patch, discuss desirability and design with other hackers @pgsql-hackers.
- Doing the necessary homework is important as giving the right and good reasons always help others to take interest in the patch.
- Getting early involvement of other community members helps in avoiding rework.

Writing Patches

- While sending the email on hackers, it is important to tell clearly (a) what problem the proposed patch is solving and (b) how it is solving it.
- Consider the below things before submitting the patch.
  - regression tests
  - documentation updates
  - if the feature is big or you are introducing some new concept, then consider adding new README or update existing README as applicable.
  - describe the effect your patch has on performance if any.

Writing Patches

- Few more things that we need before submitting the patch.
  - pg_dump support, for any new syntax or change in existing syntax.
  - does it need any change in pg_upgrade (compatibility across versions)?
- Register the patch in Open commitfest.

Important coding guidelines

- Follow the style of the adjacent code.
- Avoid unrelated white space changes.
  - git diff --check helps with that.
- Naming: We tend to use CamelCase or underscores: thisStyleIsOkay or this_is_okay_too
- Always run pgindent before submitting the patch.
- Source code formatting uses 4 column tab spacing.

Important coding guidelines

- Use C style comment (/* comment */), not C++ style (//).
- The comments should explain that rationale behind the code rather than what coding is doing.
- Limit line lengths so that the code is readable in an 80-column window.
- We sometimes go past 80 columns limit to maintain readability of the code. For ex. breaking the long error message string in arbitrary places just to obey this rule is not advisable.
- The preferred style for multi-line comment blocks is

```
/*
 * comment text begins here
 * and continues here
 */
```

Important coding guidelines – Error reporting

- All the user-facing error, log or warning messages are reported using ereport.
- The two primary elements for error reporting are (a) Severity level (DEBUG to PANIC) and (b) a primary text message.
- In addition, there are optional elements, the most common of which is an error identifier code that follows the SQL spec's SQLSTATE conventions.
- If the severity level is ERROR or higher, ereport aborts the current execution and returns the message to the user.
- For the severity level lower than ERROR, ereport just reports the message and allows the execution to proceed normally.

Important coding guidelines – Error reporting

- A typical call to ereport might look like this:

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

- There is an older function elog that is still heavily used. We typically use it for internal errors like "cannot happen" type of situations.
- The message string present in elog is not subject to translation. So this is not used for user-facing errors.

Feedback Process

- PostgreSQL development is organized into CommitFest periods.
- Four or Five per release (last year, we have five, July, Sept., Nov., Jan., March).
- This is generally the time when most of the patches got reviewed.
- Apart from committers, we need a lot of review effort during this time.
- Many-a-times developers hesitate to start with the review as they think that they might not have something useful to add. Actually, that is not true.

Feedback Process

- Even if reviewers don't know C, they can help in meaningful ways.
- Applying the patch, test the feature and report back whether it does what it is supposed to do.
- Whether the patch contains necessary regression tests or doc updates?
- If the patch indicates to have some performance impact, then doing some performance testing is useful.
- If one can't think of performance scenarios to validate, then they can ask for suggestions on the list.
- Doing these few things for a patch is also quite helpful.

Feedback Process

- The experienced reviewers can help with getting the design and code into a shape that can be committed.
- Check if the patch follows coding guidelines.
- Check for any portability issues.
- Are the comments sufficient and accurate?
- Does the patch work in corner cases (Can you make it crash in any way)?
- Is everything done in a way that fits together coherently with other features/modules?
- You can check more about reviewing the patches on the wiki.

ENTERPRISEDB

Feedback Process

- Reasons your patch might be returned
  - The fastest way to get your patch rejected is to make unrelated changes.
  - Consider how the patch would be reviewed. Do self-review before posting the patch.
  - The performance gain is claimed without a test case.
  - Did not include documentation or regression tests.
  - Failed to address earlier criticisms of this design.

Feedback Process

- During the review, both author and reviewer should update the commitfest status.
- Needs Review: You can add yourself as a reviewer of the patch. It is better to add yourself to the patches which have no other reviewer but having more people review bigger patches is certainly good too.
- Waiting on Author: After sending the review, the reviewer should change the status of the patch. This indicates that the author of the patch has to address review comments raised.
- Ready for Committer: This indicates that the patch reviewer(s) has finished the review and passed it to committer for further review/commit.
- Others: Committed, Returned with Feedback, Rejected.

How much time is required to get a patch committed?

- The time between a patch is proposed on pgsql-hackers and it got committed can vary from few days to many months depending on the patch.
- Simple patches that fix typo's or some trivial doc or code bug fix patches many a time got committed within a day or two.
- Feature or performance patches take a much longer time.
- Generally, they have to go through multiple rounds of review and revision before they get committed.
- It many-a-time also depends on how much other community members are interested in the proposed feature.

Case study for ALTER SYSTEM

- The patch was proposed on 2012-10-29 and got committed on 2013-12-18.
- It took about 14 months.
- Note that this same feature was attempted previously multiple times, but never came to conclusion.
- The feature size was 800~900 lines including doc changes.
- In this particular case, it was not the code review which took so long, rather it was primarily about the approach for implementing the feature which was discussed for a long time.

Case study for ALTER SYSTEM

- The other things which took time
  - The syntax for this command.
  - Then, we also discussed the vulnerabilities it can add to the system.
  - The primary author of the patch (myself) has changed the company in between, that doesn't have a big impact, but surely it has some impact.
  - As the approach to implementing the feature was being argued heavily, the reviewers were not clear and it took a long time for the same.

Case study for ALTER SYSTEM

- Learnings:
  - The review of feature patches sometimes has to go through multiple rounds of reviews for design, syntax, code, docs.
  - Better to discuss the design or syntax of the feature before implementation.
  - Would this patch be committed in shorter duration? maybe, but I am not sure.
  - Having patience and don't give up attitude helps in bringing such features to a conclusion.

Case study for Reduction in WAL for Update operation

- The patch was proposed on 2012-08-03 and the same got committed on 2014-03-12
- It took about 19 months.
- The committed code was about 300~400 lines.
- The idea for this work was originally posted by Simon Riggs in 2007.
- The main thing which took so much time, in this case, was to come up with an approach where we reduce the WAL by not compromising the performance.

Case study for Reduction in WAL for Update operation

- We have tried using various compression techniques including trying our own diff method, but all of the approaches have some CPU overhead.
- Finally, Heikki Linnakangas came up with an idea to find unchanged bytes in the beginning and end of the tuple which has minimal or no overhead and covers important use cases.
- Trying multiple approaches for the patch need a lot of time.
- Doing performance testing by multiple people to validate the results also require quite some time.

Case study for Reduction in WAL for Update operation

- Similar to Alter System, here also what worked was to keep trying different approaches based on community feedback.
- Of course, then there are features like Logical Replication or Parallel Query which are so big that they require multiple release cycles to be finished.

Case study for write-ahead logging support for hash indexes

- The patch was proposed on 2016-08-23 and the same got committed on 2017-03-14.
- It took about 7 months.
- The committed code was about 2100 lines.
- As compared to the previous cases, in this case, 2~7 times bigger size code got committed in less than half the time.

Case study for write-ahead logging support for hash indexes

- The main reasons why this patch didn't take as long as the previous two patches:
  - A lot of community members shown interest in this patch.
  - See commit message:
    commit c11453ce0aeaa377cbbcc9a3fc418acb94629330
    Author: Robert Haas <rhaas@postgresql.org>

    Amit Kapila, reviewed and slightly modified by me. The larger patch series of which this is a part has been reviewed and tested by Álvaro Herrera, Ashutosh Sharma, Mark Kirkwood, Jeff Janes, and Jesper Pedersen.

Case study for write-ahead logging support for hash indexes

- Some more reasons:
  - Preparatory work to make hash indexes concurrent has been already done as a separate patch prior to this work.
  - A separate mechanism for wal_cosnsistency_checker has been done separately which helped us in verifying the feature.
  - The overall approach proposed for this feature was accepted with some rework.
  - Senior community member Robert Haas was involved in the design aspect of work and Jeff Janes has done very good work in verifying this feature.

Case study for write-ahead logging support for hash indexes

- Agreement on the design and involvement of multiple community members help in moving the patch forward.
- Dividing the work in separate patches make it easier to pass through the community process.
- Responding to review comments quickly can help others engaged.

Case study for write-ahead logging support for hash indexes

- Fixing post commit complaints/bugs.
  - Tom Lane raised a complaint after the patch got committed and I provided the patch to fix it. Then Robert Haas committed the same.
  - Thomas Munro raised another complaint for which I provided a bug-fix patch. Then Robert Haas committed the same.
  - A few months later, a bug has been found by me in the implementation. I have proposed a fix for the same which is then committed by Robert Haas.

Thanks!