TÉCNICO LISBOA

PIC REPORT

Feature Description

We proposed to help enhance two subcommands, 'stor insert' and 'stor update', which both belong to the 'stor' command category. This category allows working with the in-memory SQLite database, thus these subcommands allow to insert or update information into a specified table stored in the in-memory SQLite database. To specify the target table, the users need to use the '--table-name' flag. To provide data, they must use the '--data-record' or '--update-record' flag depending on the command.

Our feature enhancement enables pipeline input support for these commands, allowing users to directly pass data without relying solely on flag parameters. Initially, we considered supporting tables as input to the pipeline. However, after consulting with a developer, we determined that for now, the commands will exclusively receive individual records from the pipeline.

Firstly, we had to modify the signature of the commands to accept a record from the pipeline, with "record" being the type of data these commands accept. We also changed the data flags to be optional parameters, as we are implementing an alternative way to pass the data.

Then, we created a function called "handle", which checks if the record is being passed through pipeline input or the respective data flag. This function ensures that only one method of passing the record is used, returning an error if both the pipeline and flag are used simultaneously.

Predicted vs actual effort

The initial estimate for implementing this feature was 200 lines of code (LoC). However, we ended up adding 253 lines and removing 135 lines. This means we not only wrote new code from scratch but also made modifications to existing code.

From what the dev said, we thought it wasn't necessary to implement unit tests since the type of commands we worked on didn't really require them.

The predicted implementation effort and debugging was 12h and 4h, respectively. However, we spent 14 hours implementing and 10 hours debugging, because our unfamiliarity with Rust and the Nushell project's specific coding style contributed to this time increase.

Lessons and Challenges

Learning Rust was our biggest challenge, as it was our first experience with this programming language. We delved into the intricacies of Rust's unique features, such as the '`Option`' and '`Result`' types, which can be particularly confusing for newcomers. These types are

fundamental to Rust's approach to error handling and data safety, but they require a different mindset compared to more traditional error handling mechanisms found in other languages.

Additionally, adapting to the coding style and conventions used in the Nushell project was another significant difficulty. Each project has its own style and set of best practices, and Nushell was no exception. We had to familiarize ourselves with their specific idioms and code structure to ensure our contribution was consistent and maintainable. This process included understanding their module organization and the overall architectural patterns employed in the project. The Nushell project uses a lot of enums in its internal data representation and there are a lot of match expressions throughout the codebase. This, combined with the need to handle a lot of edge cases, and be defensive about any errors, has led to deeply nested and complex code, very difficult to read.

Another challenge that we've encountered is the workflow of an open-source project. Sometimes it's hard to commit to an issue because, at any point, someone could submit a PR on something you've been working on for a while, making all that work and time spent feel in vain. Through this, we learned that in programming, speed really counts. Therefore, organization becomes a major factor because when you have multiple things to work on at the same time, you need to develop a way of working so your time is used efficiently and purposefully.

Documentation

As mentioned before, adapting to the code was challenging, and the project documentation didn't really help us that much in some cases. For example, the FAQ markdown hasn't been updated in six months and remains incomplete. Improved and more up-to-date documentation would greatly facilitate the process of contributing, making it easier to understand and work with the codebase of Nushell.

However, there are some pieces of documentation that are well documented, such as their Contributing markdown that really helps people that are starting with open-source projects contributions.

Overall Experience

Despite some of the limited information the documentation provides the community is very responsive and always willing to answer our questions, which was incredibly helpful. Because of this support, we are eager to continue collaborating with them. Furthermore, we enjoyed working with Rust and had fun learning their code style.

We feel like this was a good opportunity to introduce us to the business world and how to interact with people who are somewhat like co-workers. It has given us tools on how to report

our work in a formal and well-presented way, ensuring that what we are doing is taken seriously.

Improvements to the project

Something we think should be improved is the project documentation. With our now deeper understanding of the codebase, we are willing to help enhance this aspect.

Nushell still needs some improvements in certain commands and the implementation of features similar to those found in the Windows Shell, for example. However, overall, it has a highly intuitive and fully customizable interface that works in conjunction with an efficient framework, culminating with the fact that it is written in Rust, which makes this shell even more efficient.