Efficiency in Code Clone Detection: A Comparative Analysis of JavaScript and Clojure

Exploring Language Performance Using the Qualitas Corpus in Big Data Analytics Education

Hadi Saghir

Freestanding course

Summary of learnings from Big Data Module

7,5 hp

2024

Git Repository: https://github.com/Hadi-Saghir/big-data-analytics

The Intricacies and Misconceptions of Code Cloning in Software Development

In the realm of software engineering, code cloning is a topic that often garners a divided opinion. It is a practice that has been both criticized and utilized, sometimes seen as a necessary evil or a shortcut that could potentially lead to maintenance nightmares. Two insightful articles delve deep into the mechanics and the philosophy behind code cloning, providing a comprehensive overview of this controversial practice.

Understanding Code Cloning Through Diverse Detection Techniques

In their article, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," authors K. Roy, J. R. Cordy, and R. Koschke embark on a quest to dissect the complex world of code clone detection. They present a panoramic view of the existing techniques and tools that serve this purpose, such as textual, lexical, syntactic, semantic, and hybrid approaches. The article is a thorough exploration of the different angles from which one can approach clone detection, acknowledging that some methods shine brighter in certain scenarios than others due to the nuanced nature of code cloning.

The study conducted by the authors is an eye-opener, as it qualitatively measures the effectiveness, accuracy, and efficiency of the various code clone detection techniques using scenario-based evaluations. They shed light on the strengths and weaknesses of each method, which aids developers in selecting the most appropriate tool for their specific code analysis tasks. This understanding is crucial, not only for the maintenance of software quality but also for the adaptation of practices that could streamline the development process.

Reevaluating the Stigma Around Code Cloning

Kapser Cory J. and Michael W. Godfrey's paper, "'Cloning considered harmful' considered harmful: patterns of cloning in software," challenges the prevailing negative stigma associated with code cloning. The authors take a stand against the blanket judgment that code cloning is detrimental to software development. Instead, they explore the various patterns of cloning present in software and suggest that not all cloning is harmful—some patterns may even be necessary or beneficial under certain circumstances.

Their research measures the patterns and instances of code cloning across various software projects to understand its real impact on software quality and maintenance. The findings reveal that cloning introduces complexities in software maintenance, but these are not always harmful. On the contrary, they highlight that different cloning patterns require different management strategies. They detail patterns such as experimental variation, boiler-plating, templating, and forking, each with its own set of considerations and implications for software development.

A Balanced View on Code Cloning

The insights from both articles suggest a balanced view of code cloning. They underline the need for a strategic approach to managing cloning, acknowledging that while certain harmful patterns like verbatim snippets should be avoided, not all clones necessitate refactoring. The context behind cloning decisions is essential to understand, and the impact of these decisions should be evaluated case by case.

This synthesis of perspectives on code cloning provides a nuanced understanding that can serve as a guide for software developers. It equips them with the knowledge to discern when cloning might be a practical solution and when it might lead to potential issues. By understanding the diverse detection techniques and the multifaceted nature of cloning patterns, developers can make informed decisions that promote software quality and maintainability.

JavaScript and Clojure in Speed Data Analysis for Clone Detection

Utilizing the Qualitas Corpus as the foundation for analysis in the "Applied Computing and Big Data" course at Blekinge Institute of Technology, we encounter an intriguing exploration into nearly big data analytics. The assignments involve processing a significant volume of Java files to detect code clones, which is a critical task in maintaining software quality and sustainability.

From the JavaScript implementation, it is observed that processing 22,131 files in larger chunks reduced the detection of clones to 2%, a notable decrease from the 10% achieved with the default chunk size. The algorithm's performance was affected by a potential memory leak, indicated by a "Possible EventEmitter memory leak detected" warning, and a hang-up during the processing, suggesting that later files or chunks are more time-consuming due to the increasing number of iterations required by the algorithm.

On the other hand, the Clojure implementation, which processed all Java files, indicates a different approach to handling data. Using functions like ts-println and addUpdate!, the process involved timestamping and storing messages in a database, likely contributing to the monitoring of the clone detection process. The transcript summary reveals that the system identified 24,961 clones from 125,200 files and 18,211,863 chunks, with a dynamic rate of chunk generation and clone candidate identification.

The growth in chunk numbers in Clojure was not steady but varied, likely due to the complexities of reading the files and database interactions such as MD5 hashing. Interestingly, as the number of clone candidates decreased, the system became more efficient in detecting and consolidating clones, suggesting a negative correlation between the volume of candidates and the speed of clone detection.

Comparative Analysis of Language Efficiency

The difference in speed of handling the corpus between JavaScript and Clojure can be attributed to the inherent characteristics of these languages. JavaScript, with its event-driven model, faced challenges with event listener memory leaks and performance degradation as the dataset grew. Clojure, with its functional approach and efficient database operations, showed a more consistent performance, managing large datasets with variable but generally reliable speed.

Clojure’s linear time complexity for generating clone candidates indicates a scalability in processing that JavaScript struggled with due to the non-linear increase in processing time as the dataset grew. This efficiency in Clojure could be due to its immutable data structures and effective use of database indexing and hashing, which are common strengths found in functional languages optimized for big data analysis.

Conclusion

The comparative analysis of JavaScript and Clojure in processing the Qualitas Corpus for code clone detection highlights the significance of language choice in handling large datasets. JavaScript showed limitations in event-driven processing for large datasets, while Clojure's functional programming paradigm provided a more scalable and predictable performance. These findings underscore the importance of selecting the right tool for the task, especially in the context of big data analytics in software engineering education at the Blekinge Institute of Technology.