Scaling Classical Clone Detection Tools for Ultra-Large Datasets: An Exploratory Study

Jeffrey Svajlenko* Iman Keivanloo+ Chanchal K. Roy*

*Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
jes518@mail.usask.ca, croy@cs.usask.ca

+Department of Computer Science
Concordia University
Montreal, Canada
i_keiv@cse.concordia.ca

Abstract—**Detecting clones from large datasets is an interesting research topic for a number of reasons. However, building scalable clone detection tools is challenging and it is often impossible to use existing state of the art tools for such large datasets. In this research we have investigated the use of our Shuffling Framework for scaling classical clone detection tools to ultra large datasets. This framework achieves scalability on standard hardware by partitioning the dataset and shuffling the partitions over a number of detection rounds. This approach does not require modification to the subject tools, which allows their individual strengths and precisions to be captured at an acceptable loss of recall. In our study, we explored the performance and applicability of our framework for six clone detection tools. The clones found during our experiment were used to comment on the cloning habits of the global Java open-source development community.**

Index Terms—**Clone detection, scalability, large dataset**

I. Introduction

Scalable clone detection is amongst the most active topics in the clone community. One of the primary targets of such research is the creation of clone corpuses from ultra large inter-project datasets, which often contain on the order of thousands of open-source systems. However, building scalable tools is challenging and it is often impossible to use existing state of the art tools for large dataset analysis, except for emerging tools which are built for extreme scalability. Reasons for their failure may include insufficient memory, computational time requirements, or limitations in their underlying algorithms.

This research is motivated by the richness of inter-project clone corpuses for software mining experiments and applications. Corpuses may be mined to study global developer behavior or to discover the seeds of new APIs and libraries. A corpus may also be used as a basis for Internet-scale clone search [1], which has applications including API recommendation and usage support. Detection scalability is achieved using either novel scalable detection techniques, or mixing classical approaches with scalability heuristics.

In this research, we are interested in evaluating a scalability heuristic we term the shuffling framework [2]. Our technique is a nondeterministic approach, which allows classical tools (i.e., those not specifically designed for scalability) to be scaled to ultra large datasets using standard hardware without altering the detection tool. The framework achieves scalability by executing the classical tools for random partitions of the dataset, with the partition contents shuffled over a number of detection rounds.

Our goal is to allow classical tools to contribute towards inter-project clone corpuses (e.g., [1]). It is not sufficient to only consult scalable clone detectors for corpus generation as classical tools have their own unique strengths and detection characteristics. Only by consulting a variety of clone detection techniques can a comprehensive corpus be constructed.

This study extends and exploits our earlier research [2]. Here we evaluate our framework's performance for the ultra large inter-project dataset IJaDataset 2.0 [3] using a selection of classical clone detection tools, including Deckard [4], NiCad [5], iClones [6], Simian [7], SimCad [8] and CCFinderX [9]. This study reports our observations and the challenges faced in executing our framework for these tools and dataset. In order to gauge the expected performance of the framework for these tools, we also executed it for standard sized datasets which allowed us to compare clone detection with and without the framework. We developed and evaluated a heuristic for estimating framework performance when the clone output was too large to process on available hardware. We used our discovered inter-project clone facts to comment on open-source Java clone characteristics. In summary, we addressed the following research questions:

RQ#1: What is the expected performance of the shuffling framework for these selected clone detection tools?
RQ#2: What is the accuracy of our efficient heuristic for measuring the performance of the shuffling framework?
RQ#3: Is our shuffling framework successful in scaling classical detection tools to ultra large datasets?
RQ#4: What are the major characteristics of cloning in the global Java open source community?

Section 2 outlines the procedure of our shuffling framework. Section 3 overviews our experimental setup, and defines our metrics, including the heuristic (**RQ#2**). Section 4 evaluates the framework's expected performance for the selected tools (**RQ#1**). Section 5 discusses our experiences in applying our shuffling framework to an ultra large dataset (IJaDataset), and reports our observations regarding the framework's performance and applicability (**RQ#3**). In Section 6 we share our observations on cloning characteristics found in the discovered clone facts of IJaDataset (**RQ#4**). Section 7 discusses related work. Conclusions and future work are presented in Section 8.

II. The Shuffling Framework

With the shuffling framework our objective is to allow clone detection tools not designed for scalability to scale to very large datasets without modification, while using limited computational resources and achieving an acceptable overall recall. The framework executes the following procedure:

(1) The source files of the dataset are randomly partitioned into *n* equal sized subsets. Subset size is chosen as to enable the clone detection tool to handle a single subset in a single run on available hardware.
(2) Each subset is searched independently by the clone detection tool (sequentially, in parallel, or distributed).
(3) The detected clone pairs are added to a clone repository.
(4) Steps 1 through 3 are repeated for *r* rounds. Multiple rounds are required as a single round achieves limited recall as there is a high chance that cloned contents are assigned to disjoint subsets.

The framework achieves scalability by partitioning the dataset into subsets manageable by the clone detection tool. The tool's recall is recovered by repeating detection after shuffling the partition contents. The goal of this non-deterministic approach is to achieve an acceptable fraction of the tool's recall within a manageable number of rounds (*O(nr)* detection experiments).

A deterministic approach would be to partition the dataset and perform detection on every pairing of the partitions. This would achieve full recall of the detection tool, but would require $O(n^2)$ detection experiments. It would require twice the number of partitions as our non-deterministic approach as partitions are combined for detection. For applications where partial recall is acceptable, our framework is more appropriate.

The key to the use of this framework is the choice of the size of the *n* subsets. Common factors for this choice include available memory, computation time and complexity, and inherit limitations in the tool's algorithms and data structures. For additional details, see our previous work [2].

III. The Corpus, Environment, Tools and Measures

A. Corpus - IJaDataset 2.0

For our experiment we used the second version of IJaDataset, which was constructed using raw data crawled in 2012 [3]. The dataset covers source code of approximately 25,000 open source Java projects. This new version of the dataset contains up-to-date source code and is two times larger than the first version, which we used in our earlier studies [2]. The dataset is based on source files mined from SourceForge and Google Code in 2012. The crawled data includes 12 million Java files, which reduced to 3 million after filtering. The second version of this dataset includes 356 million lines of code (LOC). The dataset is publicly available [10].

Outliers. Of the 3 million files in IJaDataset, 6238 are greater than 2000 lines in length. While these make up an insignificant portion of the dataset, they may place a significant strain on clone detector performance. For this reason we consider these files *outliers* of the dataset and omitted them from the experiments.

B. Hardware

For the shuffling experiments we used standard hardware with a 2.66GHz multicore processor and 8-16GB of memory. Individual rounds were executed on independent instances of this standard hardware. Instances were provided by the Bugaboo cluster of the Western Canada Research Grid (WestGrid) and the Amazon EC2 platform. These instances do not exceed the abilities of conventional workstation-class desktops, and were exploited in order to complete this study in a limited timeframe. For particularly demanding analysis of the experiment's results, an EC2 instance with 64GB of memory was utilized.

C. Clone Detection Tools

For this study, we explored six clone detection tools. Being freely available and supporting Java source code were our major deciding factors. Table 1 summarizes our selected tools and their chosen configurations. When possible, we preferred the tools' default settings.

D. Measures

The performance of our framework is measured as *total recall*, the ratio of the clones from the target tool's *gold standard* that the framework is able to find. The gold standard is the clones the target tool finds when run as is (i.e., without our framework). For the application of the shuffling framework for r rounds and n subsets, total recall is calculated using Eq. 1. As this metric considers clone pairs, it is also referred to as *clone recall* or *clone pair recall*.

$$tr(r,n) = \frac{\sum_1^r(\sum_1^n detected\ clones)}{clones\ in\ gold} \quad (1)$$

1) Heuristic-Based Total Recall Measurement

In our experience, clone detector output may be too large for the calculation of *total recall*, even with extraordinary hardware (e.g., 244GB RAM). For this reason a heuristic was devised to estimate *total recall* using limited resources. This heuristic estimates *total recall* by measuring the ratio of the cloned fragments, rather than clone pairs, from the gold standard found by the framework. *Heuristic recall* is then achieved using Eq. 2. As this metric considers cloned fragments, it is also referred to as *cloned fragment recall* or *fragment recall*.

$$hr(r,n) = \frac{\sum_1^r(\sum_1^n detected\ cloned\ fragments)}{cloned\ fragments\ in\ gold} \quad (2)$$

TABLE I. Tool Configurations

Tool	Configuration
Deckard [4] (version 1.2.3)	Minimum fragment size of 50 tokens, and a sliding window of 5 tokens. Minimum 90% clone similarity (tree-based metric).
NiCad [5] (version 3.4)	Normalized fragment size of 10-2500 lines and minimum 70% clone similarity (line-based metric).
iClones [6] (version 0.1.2)	Minimum clone fragment size of 100 tokens and minimum cloned block size of 20 tokens.
Simian [7] (version 2.3.33)	Code fragment sizes of 6 lines or greater, no identifier or literal renaming.
SimCad [8] (version 2.1)	Detection of clone pairs of all types after consistent identifier normalizer.
CCFinder [9] (version 10.2.7.4)	Minimum fragment size of 50 tokens, with a minimum unique token type of 12.

This heuristic is based on the assumption that if two cloned fragments of a clone pair have been found by our approach, then there is a good chance the clone has also been detected, or that the clone could be recovered by applying the transitive property to all found clone pairs. For example, if fragments f1, f2 and f3 have been found in clone pairs (f1,f2) and (f2,f3) then the missed clone pair (f1,f3) can be recovered. A caveat of this approach is that while it holds true for all clones of types one and two, it may not always for type three clones.

2) Evaluation of our Heuristic-Based Recall Measure

In this study, we tested the assumptions of our heuristic-based recall measurement. We searched JDK1.7 using NiCad, Simian and Deckard both as they are and with our shuffling framework. The framework was parameterized to evaluate the dataset for 15 subsets over 30 rounds. Figure 1 compares the *total recall* and *heuristic recall* for the tools after each round. For NiCad and Simian, the transitive property was applied to recover additional clones. *Recovered recall* was then evaluated as in Eq. 1 by including the recovered clones per round as part of the tool's detected clones. *Recovered recall* was not evaluated for Deckard due to the size of its output.

As can be seen from these experiments, *heuristic recall* over estimates the *total recall*, but follows a roughly similar trend with a faster decay in growth. The *recovered recall* performance for NiCad and Simian show the correctness of the heuristic. For NiCad the *recovered recall* approximately matches the *heuristic recall*. For Simian the *recovered recall* approaches *heuristic recall* after half the rounds have been executed. This shows us our heuristic is effective in estimating the recall of our shuffling framework (**RQ#2**). Note that while the transitive property is not perfect for type 3 clones, false positives "recovered" by naïve application of the transitive property does not affect recall measurement in this study. The recovery method has not been efficiently implemented for the shuffling framework, and is therefore used only with this study. We hope to integrate an efficient version in future work.

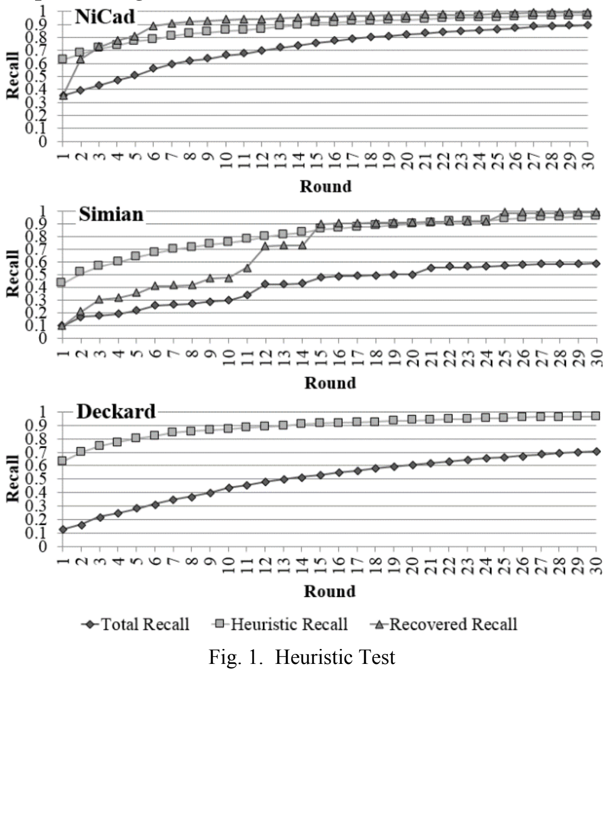

Fig. 1. Heuristic Test

IV. Preliminary Experiments

Before starting the main study, we used the shuffling framework to evaluate two regular size (i.e., small enough to evaluate gold standard) subjects systems. The goal of this experiment is to observe the expected performance of the framework for the six selected tools (**RQ#1**). We chose ArgoUML (190KLOC - 1845 files) and JDK1.7 (900KLOC - 6916 files) as our regular sized systems. The framework was parameterized for 15 random subsets and 30 detection rounds.

The framework's *total recall* performance for each tool's detection of ArgoUML is shown in Fig. 2 and of JDK1.7 in Fig. 3. The legends of the graphs specify the gold standard size (number of clones) for each tool. The framework performed very well with NiCad, iClones, and CCFinderX, obtaining a high *total recall* after 30 rounds. It struggled more for Deckard, and performed poorly with Simian for JDK1.7. *Total recall* started and ended lower for JDK1.7, but increased faster than for ArgoUML, likely due to the differences in the sizes of the two systems (and gold standards). CCFinderX is omitted form the JDK1.7 experiment due to failure during detection.

An observation from this experiment is that generally the larger the gold standard the lower the *total recall* obtained by the framework across the same number of rounds and subsets. This is seen here for both variation in detection tools and subject system size. The exception being Simian, for which the framework achieves a lower *total recall* than for tools with

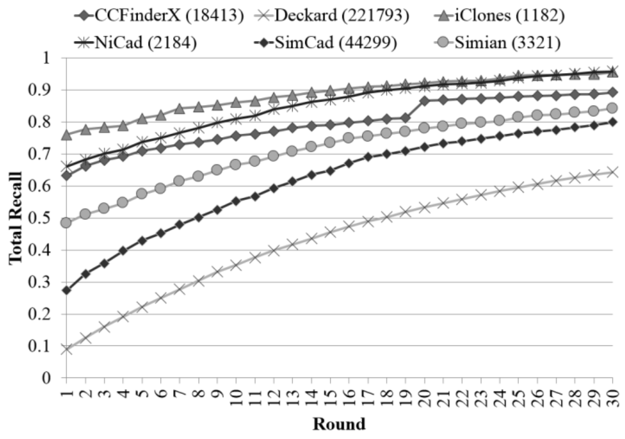

Fig. 2. Preliminary Experiment – ArgoUML

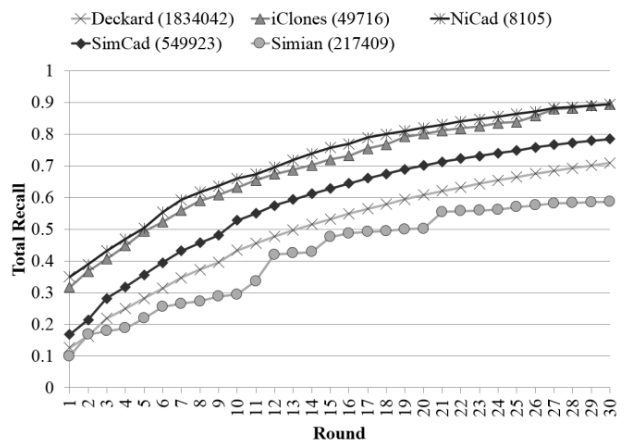

Fig. 3. Preliminary Experiment – JDK1.7

larger gold standards. Perhaps Simian has better precision for smaller datasets, and is therefore not finding the false positives in the gold standard, leading to a lowered *total recall*.

These results indicate that the framework can achieve a fair recall (>70%) for any clone detection tool given an acceptable number of rounds. The plots here show the expected framework performance for the tools, which answers **RQ#1**.

V. The Main Experiment With IJaDataset

This is the primary experiment of this research. The clone detection tools were used to evaluate the ultra large IJaDataset. The experiment was used to evaluate the performance and feasibility of the shuffling framework for evaluating an ultra large dataset using classical clone detection tools (**RQ#3**).

Of the original six detection tools only Simian, NiCad and Deckard were used successfully for this experiment. CCFinderX, iClones, and SimCad were omitted due to compatibility issues with the dataset. Table II summarizes the shuffling experiments performed.

A. Simian

Setup. Simian was chosen for this experiment as it does not encounter scalability issues with a dataset as large as IJaDataset. By evaluating the dataset using Simian both as is (gold standard) and by the shuffling framework, we were able to evaluate our technique's performance for very large datasets. Simian's gold standard was created using an Amazon EC2 instance with 68GB of RAM. For evaluation with the shuffling framework, a subset size of 50,000 files was chosen (58 subsets). Simian's fast execution let us choose 30 rounds as sufficient to demonstrate the shuffling framework. Subset generation and round detection took approximately 8-12 and 4-10 hours per round, respectively.

Analysis. Since Simian's gold standard is extremely large (300 billion clone pairs) *total recall* was estimated using the heuristic. *Heuristic recall* is shown in Fig. 4, with 70% of the cloned fragments in the gold standard found by the framework after 30 rounds. According to the study of the heuristic effects (Section 3-D-2), *total recall* for Simian should be somewhat less than *heuristic recall*, but with a faster growth. It also showed that *recovered recall* quickly approached *heuristic recall* with Simian once *heuristic recall* reached 70-80%. Simian has achieved an acceptable recall for cloned fragments within the 30 rounds, and the heuristic study suggests that the recovery method would allow it to achieve a similar recall of clone pairs, perhaps requiring 5-10 additional shuffling rounds.

While the heuristic is a worthy approximation of *clone recall* by the framework, it is still desirable to measure *total recall*, which necessitated a reduction in Simian's output. Investigation into the characteristics of Simian's gold standard

TABLE II. Summary of the IJaDataset Clone Detection Experiments

Tool	Hardware	Subset size (#files)	#Sets	#Rounds	Time Total Proc. Time (hours)
Deckard	24GB	10K	289	10	~1440
NiCad	12GB	10K	289	20	~760
Simian	12GB	50K	58	30	~510

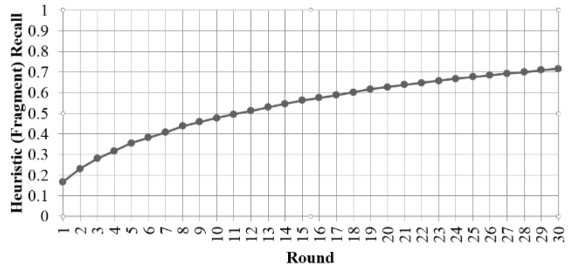

Fig. 4. Simian Heuristic-based Recall (Clone Fragment Recall)

found that 99.99% of Simian's clones came from clone classes greater than 100 fragments in size. Manual investigation into these clone classes revealed that Simian suffered from what we termed the *sliding effect*; it reported some extremely large clone classes containing the same fragment(s) repeated numerous times with small offsets in line numbers. These clone classes generate an extreme number of self (overlapping) clones and represent a significant threat to Simian's precision. We therefore reduced Simian's output size by trimming clone classes over a certain maximum size.

Figure 5 shows our framework's *total recall* using Simian for various maximum clone class sizes up to 100 fragments (limitation of our hardware). The legend of this figure specifies the maximum class size considered with the gold standard's size in parenthesis. *Total recall* was higher and increased faster for lower maximum clone class size. This suggests that the framework works best for specialized clone detection (i.e., focusing on detecting interesting/unique clones rather than all clones). This is due to larger clone classes requiring more rounds to be completely found as each fragment in the class must be shuffled into the same partition as each of the rest at least once. For the smaller class sizes a respectable *total recall* was achievable within 30 rounds (2: 52%, 5: 44%, 10: 40%). While still low, the *total recall* in each case increases nearly linearly, with very little decay in slope. Additional rounds could bring these to an acceptable level. As can be seen, a 7-10% increase in *total recall* is gained per additional 10 rounds. The recovery method would also help boost *total recall* achieved. We expect the shuffling framework may perform better for other tools, as our preliminary study found that the framework performed worst for Simian (Fig. 3, JDK1.7).

Figure 6 shows *heuristic recall* for the same trimmed output. As can be seen, the shuffling framework is finding the cloned fragments very fast, with 52-62% *heuristic recall* after only 30 rounds for each group. *Heuristic recall* increases faster for larger maximum clone class size, meaning that the fragments in large clone classes are more easily found. This is

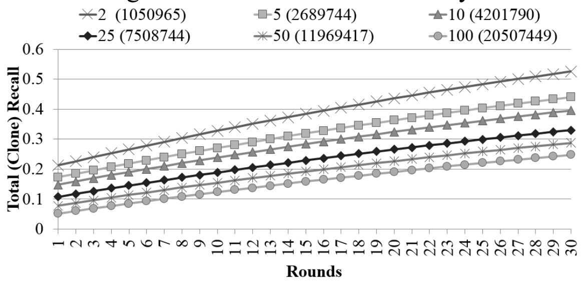

Fig. 5. Simian Total Recall for Maximum Class Size Trimmed Output

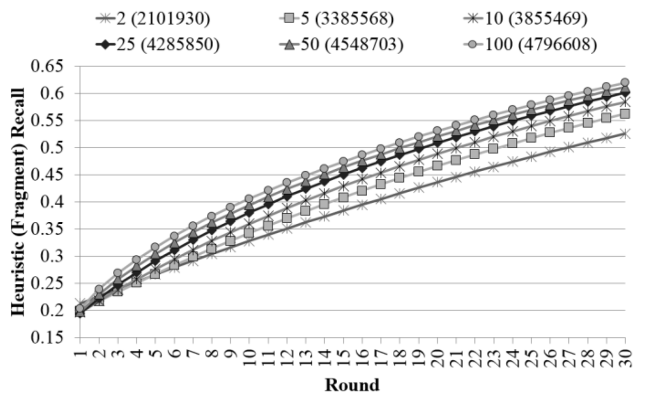

Fig. 6. Simian Heuristic-based Recall for Max. Class Size Trimmed Output

expected as fragments in large clone classes have a higher chance of being shuffled into a partition with another fragment from the clone class. This suggests that the recovery method may work especially well for the clones of large clone classes. This is particularly beneficial as it was for these clones that the framework had a slower increase in *total recall* (Fig. 5).

B. NiCad

Setup. NiCad was included in this experiment for its ability to restrict clone detection to function clones. This is especially beneficial to detection in large datasets where the number of line level clones may be too large to process. Function clones are fewer, and more likely to be interesting as they occur at a higher software design level.

Through experimentation, it was found that NiCad could safely handle datasets of 10,000 files. It failed with larger input due to hard coded limits in the sizes of its internal data structures. These limits appear to act as an early warning to guide unwary users away from scalability issues.

Based on these observations a subset size of 10,000 files was chosen for running the shuffling framework (258 subsets). As the framework achieved better *total recall* with NiCad than with Simian in the preliminary experiments and previous work [2], 20 rounds was deemed sufficient for demonstration of the framework. Subset generation and detection took 7-15 and 23-31 hours per round respectively (shared computing resource).

Analysis. Creating a gold standard for NiCad was not possible so we could not evaluate *total recall*. Instead we investigated the growth of the number of unique clones and cloned fragments found after each successive round of our framework with NiCad. This information is plotted in Fig. 7. In total 5.66 million unique clone pairs containing 875 thousand unique cloned code fragments were found.

The growth of unique reported *clone pairs* (Fig. 7 diamond-line) is roughly linear across the twenty rounds, with no significant decay in its slope. This is due to the large number of subsets created from the dataset, which was a requirement due to NiCad's scalability limits. More rounds would be required to see the growth begin to decay. In contrast, the growth of cloned fragments (Fig. 7 square-line) decays across the rounds significantly. These two facts suggest that most of the cloned fragments are being found quickly, but that the clone relationships between them are still being detected. Applying the transitive clone recovery technique would be a good way to recover some of the remaining clones without executing further rounds. As seen in the heuristic study (Section 3-D-2), clone recovery was very successful for NiCad.

C. Deckard

Setup. Experimentation found that Deckard worked for our approach with a subset size of 50,000 files, and could possibly work for larger subsets up to the entire dataset (untested). However, its execution for large sizes exceeded time constraints, so a subset size of 10,000 files was used to match NiCad (289 subsets). As Deckard has a lengthy execution time, the shuffling framework was executed over only 10 rounds for this demonstration. Detection was ran on Amazon EC2 and took approximately 5-7 days per round. We attempted to run the remaining 10 rounds on Westgrid, but found they stalled partway through the execution without error or termination.

Caveat. One disadvantage of Deckard is that it only supports up to Java 1.4 syntax. Its documentation specifies that it is able to skip unsupported syntax without error. In our experience, it found plenty of clones despite this limitation.

Analysis. Creating a gold standard for Deckard was not possible due to the computation time required, so we could not investigate *total recall*. Instead we investigated the detection growth across the shuffling rounds as we did for NiCad.

Figure 8 shows the growth of the number of unique detected cloned fragments as measured by the heuristic. As can be seen, the growth of detected cloned fragments decays over the shuffling rounds. Unfortunately, we could not measure the detected clone pairs across the rounds due to the size of Deckard's output. We can infer from NiCad's and Simian's results it would likely be increasing linearly over these rounds.

In order to confirm our inference, we measured found clone pairs and fragments on a reduction of Deckard's output. We reduced the output sized by considering only reported clone classes with a maximum size of 10 fragments (limitation of our hardware). The growth of detected clones and fragments for this reduced output is shown in Fig. 9. As expected we found very similar results to NiCad. The detected clones increases roughly linearly, while the detected fragments increases with significant decay. Again this suggests that the code fragments are found early compared to clone pairs, and that the recovery method would be useful in boosting the found clone pairs.

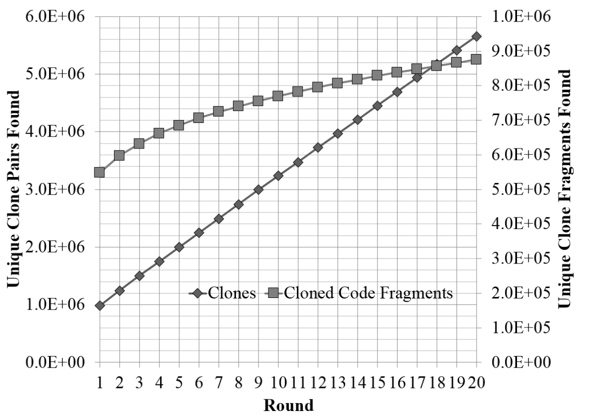

Fig. 7. Growth of NiCad's Found Clones and Cloned Fragments

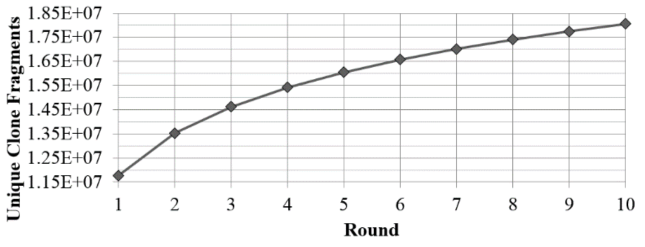

Fig. 8. Growth of Deckard's Found Cloned Fragments

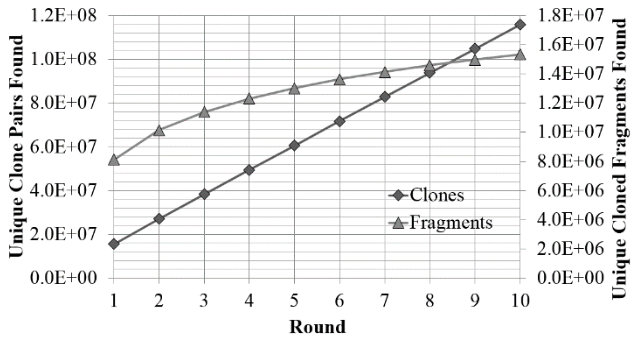

Fig. 9. Deckard's Clone and Fragment Detection Growth for Reduced Output

D. Other Tools – SimCad, iClones, CCFinderX

Our intention was to include SimCad, iClones and CCFinder in the main experiment as they showed promise in the preliminary experiment. During evaluation of a sample from the dataset, these tools terminated with an error. CCFinderX failed silently, while iClones and SimCad reported encountering an invalid Unicode character. This problem does not indicate scalability issues with these tools, or with our framework. Communication with the iClones developers revealed this has been fixed in a development branch. This problem was also reported to the SimCad developer, and has since been corrected. Strict time constraints prevented us from re-integrating these tools into the experiment. We plan on investigating these tools with the framework in future work.

E. Conclusions of the Main Experiment

From these experiments, we found that the clones found by the framework increased nearly linearly, with a slight decay in slope, across the rounds. This shows that additional rounds would continue to see a healthy increase in found cloned pairs, and thus an increased *total recall*. For Simian and considering only smaller clone classes (2-100 fragments) 25-52% *total recall* was achieved over 30 rounds, with a (decaying) continued increase of 7-10% per 10 rounds (Fig. 5). Further detection rounds could bring *total recall* to an acceptable value.

However, the framework was able to find the clone fragments much faster. For each tool, the growth of found cloned fragments decayed rapidly across the rounds. Simian's results showed that this was due to a majority of the cloned fragments having been found (Fig. 4, Fig. 6). 70% were found within 30 rounds (52-64% specifically for small clone classes).

These findings suggest that our framework finds most of the cloned fragments in few rounds, but may require a large number of rounds to find all of the clone relationships between them. This suggests that a transitive-based clone recovery process could improve total recall achieved. This is supported by our heuristic study (Section 3-D-2) which showed that a strong *heuristic (clone fragment) recall* can be translated into a strong *total (clone) recall* by transitive recovery. Implementing this recovery process is therefore a priority for our future work.

From our experiment, we conclude that the shuffling framework is successful in scaling classical clone detection tools to ultra large datasets (**RQ#3**). It is best suited for applications which accept partial detection tool recall as sufficient. For example, when building a comprehensive inter-project clone corpus (e.g., for IJaDataset) using both classical and scalable detection tools, a 60-80% partial recall using our framework is likely sufficient to ensure the clone corpus benefits from the diverse strengths and detection characteristics of these classical tools.

The framework is also very suitable for applications which only require knowledge of the cloned fragments within an ultra large dataset, and not the pairs. Given that we encountered scalability limits (memory and time) in processing the clone pairs found by this experiment, it is likely that studies on inter-project clone corpuses of similar scale may need to be done on cloned fragments. Analyzing clone pairs found may require extraordinary hardware and long computation time.

VI. Characteristics of the Detected Clones

Here we report our observations of the clones discovered in the IJaDataset experiment, which represents cloning behavior in the Java open source community (**RQ#4**).

Clone Report Size. Clone report size was measured for each tool as the number of unique clone pairs and cloned fragments found (Table III). Due to the limits of our hardware we were only able to count the clone pairs Deckard found in its first round (1.4 billion). These results show that line level and token level clone detection tools report significantly more clones than function-level detection tools. The implication is that higher granularity detection techniques may be more desirable for creating an inter-project clone corpus as their smaller reports will be easier to process and analyze.

Clone Class Statistics. Table IV summarizes our measurement of the size of clone classes reported by the tools. For NiCad clone classes were discovered by clustering its clone pairs using the transitive property. Only the first round of Deckard was considered as it was not possible to consolidate its classes across its rounds on our hardware. The primary observation here is that while small clone classes (2-5 fragments) are overwhelmingly the most common of the reported clone classes, they make up an extreme minority of the total reported clone pairs. This indicates that inter-project

TABLE III. The Clone Dataset and Clone Network Characteristics

Property	NiCad (20R)	Simian (Gold)	Deckard (10R)
#Unique Clones	5.66 Million	298 Billion	1.4 Billion (1st round)
#Unique Fragments	876 Thousand	10.8 Million	1.81 Million
Avg.CloneSize (LOC)	21.6	40.6	24.5
ModeCloneSize (LOC)	11	15	19

TABLE IV. CLONE CLASS SIZE STATISTICS FOR NICAD AND SIMIAN

Class Size	NiCad (20R)		Simian (Gold)		Deckard (1R)	
	Freq.	*% clones*	*Freq.*	*% clones*	*Freq.*	*% clones*
2-5	211285	0.32	1431918	0.00091	4081215	0.66
6-10	7620	0.15	64083	0.00051	383954	0.68
11-20	2394	0.21	23112	0.00077	208870	1.54
21-50	1663	0.75	12064	0.0019	192832	7.3
51-100	888	1.9	3615	0.0030	31828	4.7
101+	659	97	2600	99.99	9508	85

clone corpuses would benefit from clustering as a pre-processing measure to reduce data size and processing costs.

Clone Size. Figure 10 shows the distribution of cloned fragment sizes in IJaDataset for the tools plotted logarithmically. The majority of the clones reported are in the 0-25 and 26-50 LOC ranges. Clones of smaller size typically occur at an order of magnitude or lesser frequency. This is as expected as cloning typically occurs at the function or code block level, not at the class level in Java source code. Average and mode fragment sizes for the tools are shown in Table III.

VII. RELATED WORK

Scalable clone detection research can be summarized as five unique approaches: (1) deterministic novel general purpose detection e.g., [11], (2) deterministic novel domain-specific approaches e.g., [12], (3) deterministic approaches for achieving scalability using an available clone detection tool as is e.g., [13], (4) deterministic approaches for achieving scalability by altering available tools e.g., [14], and (5) *nondeterministic* approaches for scalability without altering e.g., [2]. A variety of use cases can be addressed using each family based on their unique features. Which technique should be employed depends on the intended application.

There are few recent and similar studies to our research in the literature. Ishihara et al. [12] exploited the inter-project scalable clone detection to locate commonly used functionalities over 13K open source projects in order to generate the seed for future APIs and libraries. Schwarz et al. [15] studied cloning between ~3K Smalltalk projects to deploy a database of clones which can be queried. Ossher et al. [16] observed the cloning at file level using coarse-grained clone detection heuristics. Common to all these studies, the detection approach is customized and optimized considering the research objectives and requirements (i.e., scalability). This is contrary to our research where we tried to generate and study a clone dataset using available clone detection tools by copping with the scalability issue without altering the tools.

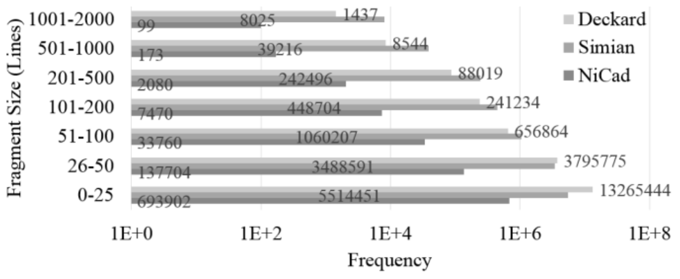

Fig. 10. Clone Size Frequency for IJaDataset

VIII. CONCLUSION AND FUTURE WORK

In this research we have demonstrated that the shuffling framework can be effectively (**RQ#3)** used to scale existing clone detection tools to ultra large datasets. As future work we plan to (1) expand our shuffling framework experiment to further tools, (2) investigate clone recovery methods to increase *total recall*, and (3) use the shuffling framework to contribute toward a validated comprehensive clone corpus for IJaDataset.

ACKNOWLEDGEMENTS

We thank the developers of the tools used in this study, as well as Dr. Rilling and co-authors of our earlier publications on the shuffling framework [2] and IJaDataset [3].

REFERENCES

[1] I. Keivanloo, C. Forbes, and J. Rilling, “Similarity search plug-in: Clone detection meets internet-scale code search”, Proc. ICSE SUITE, 2012, pp. 21-22.

[2] I. Keivanloo, C. K. Roy, J. Rilling, and P. Charland, “Shuffling and randomization for scalable source code clone detection”, Proc. IWSC, 2012, pp. 82-83.

[3] I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, and J. Rilling, “A linked data platform for mining software repositories”, Proc. MSR, 2012, pp.32-35.

[4] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones”, Proc. ICSE, 2007, pp. 96-105.

[5] C. K. Roy and J. R. Cordy, “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization”, Proc. ICPC, 2008, pp.172-181.

[6] N. Göde and R. Koschke, “Incremental clone detection”, Proc. CSMR, 2009, pp. 219-228.

[7] Simian, http://www.harukizaemon.com/simian/, Jan. 2013.

[8] S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, “On the effectiveness of Simhash for detecting near-miss clones in large scale software systems”, Proc. WCRE, 2011, pp. 13-22.

[9] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multi-linguistic token-based code clone detection system for large scale source code”, IEEE Trans. Softw. Eng., 2002, pp. 654-670.

[10] SeClone IJaDataset, http://secold.org/projects/seclone, Jan 2013.

[11] R. Koschke, “Large-scale Inter-system clone detection using suffix trees”, Proc. CSMR, 2012, pp. 309-318.

[12] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, “Inter-project functional clone detection toward building libraries - An empirical study on 13,000 projects”, Proc. WCRE, 2012, pp. 387-391.

[13] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, “Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder”, Proc. ICSE, 2007, pp. 106-115.

[14] H. Sajnani, J. Ossher, and C. Lopes, “Parallel code clone detection using MapReduce”, Proc. ICPC, 2012, pp. 261-262.

[15] N. Schwarz, M. Lungu, and R. Robbes, “On how often code is cloned across repositories”, Proc. ICSE, 2012, pp. 1289-1292.

[16] J. Ossher, H. Sajnani, and C. Lopes, “File cloning in open source Java projects: The good, the bad, and the ugly”, Proc. ICSM, 2011, pp. 283-292.