Clustering Large-Scale, Distributed Software Component Repositories

Marcos Paulo Paixão, Leila Silva
Computing Department
Federal University of Sergipe
São Cristóvão, SE, Brazil
marcospsp@dcomp.ufs.br, leila@ufs.br

Gledson Elias
Informatics Department
Federal University of Paraíba
João Pessoa, PB, Brazil
gledson@di.ufpb.br

Abstract — **In software component repositories, search engines have to deal with challenges related to storage space requirements for indexing semi-structured data models, which are adopted for representing syntactic and semantic features of software assets. In such a context, clustering techniques seem to be attractive for reducing the number of assets in a repository, and so, the size of index files. Accordingly, this paper proposes and evaluates a distributed clustering approach for large-scale, distributed software component repositories. Based on experiments, outcomes indicate relevant gains in storage space requirements for index files.**

Keywords-clustering techniques, indexing techniques, search engines, software component repositories;

I. INTRODUCTION

Software component repositories have to handle metadata for describing stored software assets, providing information employed by search engines for indexing them [1]. As endorsed by Vitharana [2], component description models can adopt high level concepts for describing component metadata, making possible to express syntactic and semantic features, and so, facilitating developers to search, select and retrieve assets. In practice, currently available component description models [3][4], have adopted approaches based on semi-structured data, more specifically XML, allowing structural relationships among elements to aggregate semantic to textual values.

Several proposals exist for indexing semi-structured data [5][6][7]. Despite their contributions, existing techniques still suffer from problems related to storage space requirements, processing time and precision level of queries. For instance, Brito et al. [7] proposes an indexing technique based on semi-structured data, which is precise and efficient in terms of query processing time, but suffer from problems related to storage space requirements for index files. Thus, in the context of large-scale software component repositories, it is a challenge to design indexing techniques that minimize storage space requirements, but without excessively impacting on query processing time and precision level.

In such a context, an interesting insight for optimizing the storage space required by index files is to construct a clustered repository, in which clusters (groups) of similar software assets are identified by applying a clustering heuristic, according to defined similarity criteria. Then, representative assets of the clusters can be generated for defining the clustered repository. Thus, only the reduced set of representative assets in the clustered repository needs to be indexed, instead of all assets in the original repository. As a result, storage space requirements can be notably reduced.

Motivated by such an insight, in [8], we have already proposed a clustering approach for X-ARM based software component repositories, which constructs a clustered repository, reducing the number of assets, and so, optimizing the storage space requirements. Despite the excellent gains in storage space requirements, the clustering approach proposed in [8] operates in local and centralized repositories only.

Based on Seacord [9], local and centralized repositories lead to limited accessibility and scalability. Besides, according to Frakes [10], a large quantity and variety of components can increase software reuse. Thus, scalability issues associated with centralized repositories obviously also imply in low reusability levels. As a consequence, large-scale, distributed software component repositories have been proposed as an infrastructure for minimizing problems related to limited accessibility, scalability and also reusability provided by centralized repositories [11][12].

Accordingly, herein, we have evolved the centralized clustering approach, already proposed in [8], to a distributed clustering approach for large-scale, distributed software component repositories, defined as a collection of connected, integrated storage units that reside throughout independent nodes of the repository in a network like the internet, for instance. The proposed approach is structured in two stages. The first stage is responsible for clustering each storage unit of the distributed repository, generating a set of clustered units. Then, the second stage puts together all clustered units and once again applies the clustering algorithm for constructing the whole clustered repository.

This paper is organized as follows. Section II describes related techniques, evincing the contribution of evolving the centralized cluttering approach. Section II reviews the adopted component description model, called X-ARM. Then, Section IV presents the proposed distributed clustering approach. After that, outcomes observed in experiments are presented in Section V. In conclusion, Section VI presents final remarks and delineates future work.

II. RELATED TECHNIQUES

Data clustering is a NP-hard problem and several heuristics have been proposed to group similar data. Xu and Wunsch [13] present a relevant survey of the research field. Among existing techniques, the hierarchical and the

 ISBN: 978-1-61208-185-4

K-means algorithms are applied in several domains [14]. In software engineering, clustering techniques have been used in several problems, such as: software maintenance [15], modularization [16], reverse engineering [17], software evolution [18] and software requirements [19].

Although clustering techniques are applied in several problems of software engineering, for the best knowledge of the authors, we are the first research group that adopts such techniques in the context of indexing software component repositories, proposing a centralized clustering approach for reducing the storage space requirements in local and centralized software component repositories [8].

As an evolved version of the previous centralized approach, the differential and so the contribution of the evolved approach proposed herein is twofold. First, the proposal of a new heuristic, also based on the hierarchical algorithm, but adopting a divide and conquer strategy that can be adopted for large-scale, distributed software component repositories. The second contribution is the consolidation of the similarity metric, initially proposed in the centralized version of the clustering approach.

The proposed heuristic has been validated by considering several thresholds for guiding the composition of component groups and also different repository sizes. Moreover, the heuristic was integrated as part of the indexing system adopted in [7], allowing quality evaluation for queries.

III. THE X-ARM MODEL

The proposed approach explores the X-ARM description model, which is a XML based semi-structured data model for describing several types of software assets [4], which can be produced in CBD (Component-Based Development) processes, proving the required semantic for representing their relationships. As illustrated in Fig. 1, X-ARM describes component and interface specifications, as well as component implementations.

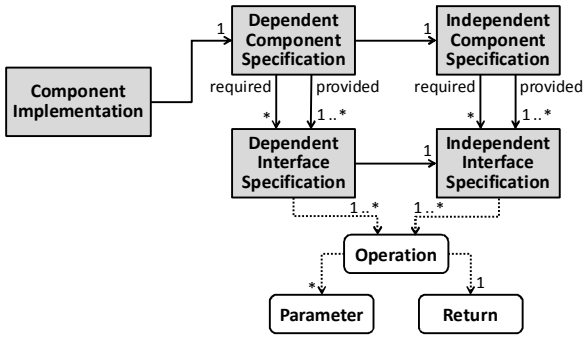

Figure 1. Relationships between assets.

Component and interface specifications can be described as independent or dependent of component models. Independent specifications abstract features and properties of component models, such as CCM, JavaBeans, EJB and Web Services. In turn, dependent specifications ought to consider features and properties related to component models.

In X-ARM, both dependent and independent interface specifications are described as a set of operations. Each operation has a name, a set of input or output parameters and a return value. In CBD processes, dependent interfaces must be in conformance with their independent counterparts. So, in Fig. 1, dependent interfaces refer to their respective independent interfaces.

Dependent and independent component specifications have a set of provided and required interface specifications. However, note that dependent and independent component specifications refer to dependent and independent interface specifications, respectively. In CBD processes, dependent components must be in conformance with their respective independent counterparts. Thus, dependent components refer to their respective independent components.

Similarly, in CBD processes, component implementations must be in conformance with their respective dependent component specifications. So, in Fig. 1, note that component implementations must refer to their correspondent dependent component specifications. Besides, for each dependent component specification, several component implementations can be realized.

As an X-ARM example, Fig. 2 illustrates a fragment of a dependent component specification. Line 1 is the asset header, which has the asset identifier *(id)*. Lines 2 to 4 refer to its respective independent component specification. Then, lines 5 to 14 refer to all provided dependent interface specifications. Although not illustrated in Fig. 2, required interfaces can also be similarly specified.

```
01 <asset name="dependentCompSpec-X"
          id="compose.depenpentCompSpec-X-1.0-beta">
02   <model-dependency>
03     <related-asset name="independentCompSpec-Z"
                      id="compose.independentCompSpec-Z-1.0-stable"
                      relationship-type="independentComponentSpec"/>
04   </model-dependency>
05   <component-specification>
06     <interface>
07       <provided>
08         <related-asset name="dependentInterface-A"
                          id="compose.dependentIntSpec-A-2.0-stable"
                          relationship-type="dependentInterfaceSpec"/>
09       </provided>
10       <provided>
11         <related-asset name="dependentInterface-B"
                          id="compose.dependentIntSpec-B-3.0-stable"
                          relationship-type="dependentInterfaceSpec"/>
12       </provided>
13     </interface>
14   </component-specification>
15 </asset>
```

Figure 2. Dependent component specification in X-ARM.

IV. A DISTRIBUTED CLUSTERING APPROACH

The task of indexing repositories based on semi-structured data is a well-known relevant issue [5][6][7]. One of the major challenges is to provide an indexing mechanism that reduces storage space requirements, but without excessively impacting on query processing time and precision level. As mentioned, we have been investigating clustering techniques as a mean to handle such a challenge.

As a result of our investigations, we propose herein a distributed clustering approach for optimizing storage space requirements in large-scale, distributed software component repositories. To do that, the approach constructs a clustered

 ISBN: 978-1-61208-185-4

repository in which each asset of the original repository belongs to a group (cluster), formed by applying a clustering heuristic, according to similarity criteria detailed afterwards. The clustered repository is composed only by representative assets of the identified clusters. Thus, instead of indexing the original repository, the search engine needs only to index the reduced set of representative assets in the clustered repository, each one referring to its respective original assets.

Clustering techniques [14] consist of three basic phases: *(i)* extraction of relevant features that express the behavior of the elements to be clustered; *(ii)* definition of a similarity metric, used to compare elements; and *(iii)* application of a clustering algorithm in order to construct the clusters. For extracting features, it is identified which information is relevant to capture the behavior of the elements to cluster and how this information is quantified. The result is a vector of relevant features, called *attribute vector*. The similarity metric expresses, in quantitative terms, the similarity between elements. In general, a function is defined and the Euclidean distance [14] between two elements is one of the more common adopted metrics. Finally, the clustering algorithm is a heuristic that generates groups of similar elements, according to the adopted similarity metric.

It must be emphasized that both the centralized approach proposed in [8] and the distributed approach proposed herein adopt the same relevant features and similarity metrics. Thus, such features and metrics are briefly reviewed herein.

A. Relevant Features and Similarity Metrics

The proposed approach considers five types of X-ARM assets and the clustering technique is applied separately for each one, which has a distinct attribute vector for representing its relevant features. Thus, similarity metrics are different for each type of asset.

The similarity between two assets a and b is quantified by an integer number, called *distance*. To avoid negative distances, the initial default distance d_i is 300. The similarity criterion is applied and this value may decrease, in such a way that assets are more similar when the final distance $d_f(a,b)$ approximates to zero. So, for each type of asset, the similarity between two assets a and b is defined by (1), in which the term $s(a,b)$ represents a factor derived based on the similarity criterion for the respective type of asset.

$$d_f(a,b) = d_i - s(a,b) \quad (1)$$

The relevant features of an independent interface specification are its defined operations, considering their names, input and output parameters and return values. Thus, independent interfaces are similar when they have in common a considerable subset of operations. In this case, the term $s(a,b)$ is defined by $op(a,b) \times 300$, where $op(a,b)$ represents the percentage of common operations provided by both interfaces. Note that the similarity criterion takes into account syntactic features only. Thus, when operations are semantically similar but syntactically different, the proposed approach cannot detect similarity, reducing its effectiveness.

Taking into account dependent interface specifications, the relevant features are the referenced independent interface specification together with their operations. Hence, dependent interface specifications are similar when they refer to the same independent interface specification or have in common a considerable subset of defined operations. Accordingly, $s(a,b)$ is expressed as the sum of two terms: $l(a,b)$ and $op(a,b) \times 100$. The term $l(a,b) = 200$ if both assets refer to the same independent interface specification; otherwise it is 0. In turn, the term $op(a,b)$ is the ratio of common operations provided by both interfaces.

In relation to independent component specifications, the relevant features are the set of provided independent interface specifications. So, independent component specifications are similar when they have in common a considerable subset of provided independent interface specifications. Due that, the term $s(a,b)$ is given by $p(a,b) \times 300$, where $p(a,b)$ is the percentage of common independent interfaces provided by both assets.

For a dependent component specification, the relevant features are its referenced independent component specification, as well as its set of provided dependent interface specifications. Therefore, dependent component specifications are similar when they refer to the same independent component specification or have in common a subset of provided dependent interfaces. As a result, $s(a,b)$ is the sum of two terms: $k(a,b)$ and $p(a,b) \times 100$. The term $k(a,b) = 200$ if both assets refer to the same independent component specification; otherwise it is 0. In turn, the term $p(a,b)$ is the ratio of common dependent interface specifications provided by both assets.

Finally, for a component implementation, the relevant feature is its referenced dependent component specification. Hence, different implementations of the same dependent component specification are considered similar. Due that, the term $s(a,b) = 300$ if both assets refer to the same dependent component specification; otherwise it is 0.

B. Clustering Algorithm

The proposed clustering algorithm has been designed to be applied in distributed software component repositories, like X-CORE [11], defined as a collection of connected, integrated storage units, each one located in different independent nodes, which are dispersed throughout a network like the internet, for instance.

The clustering algorithm has two stages. The first stage adopts a divide and conquer strategy, which is responsible for separately clustering each storage unit of the distributed repository, generating a set of independent clustered units. Subsequently, the second stage joins together all clustered units and generates the whole clustered repository by adopting a pair-wised clustering scan. In order to reduce processing time, the first stage must be concurrently performed in all storage units, and the second stage ought to be concurrently performed in a single node using a multi-threaded, pair-wised clustering scan. In the following both stages are described in more details.

In the first stage, which is executed in each storage unit, initially, assets are randomly chosen from the storage unit and kept in primary memory. It is suggested to exhaust memory capacity with this operation. Next, but still in the first stage, the classical hierarchical clustering algorithm [14]

 ISBN: 978-1-61208-185-4

is applied to these assets. Initially, each asset is considered a cluster. Then, the algorithm groups successively the two nearest clusters, until the distance between clusters is greater than an established threshold, specified by the user. The algorithm considers the similarity metric described previously to compute the distance. The combined cluster is considered a representative asset of the joined clusters. For each type of asset, the representative asset includes the relevant features for the similarity metric and also references to the joined assets. At the end of the iteration, a directory containing all formed representative assets (clusters) is saved in secondary memory. Then, in a new iteration, another set of assets, remaining in the original storage unit, is arbitrarily chosen and the process repeats. At the end of the first stage, several directories of representative assets have been constructed, one for each iteration of the algorithm. Fig. 3 illustrates the main steps of the first stage: *(a)* assets are randomly selected from the original storage unit; *(b)* clusters composed of similar assets are constructed by applying the hierarchical clustering algorithm; *(c)* representative assets are created in the respective directory for representing each cluster; and *(d)* a clustered unit is defined based on directories created in each iteration.

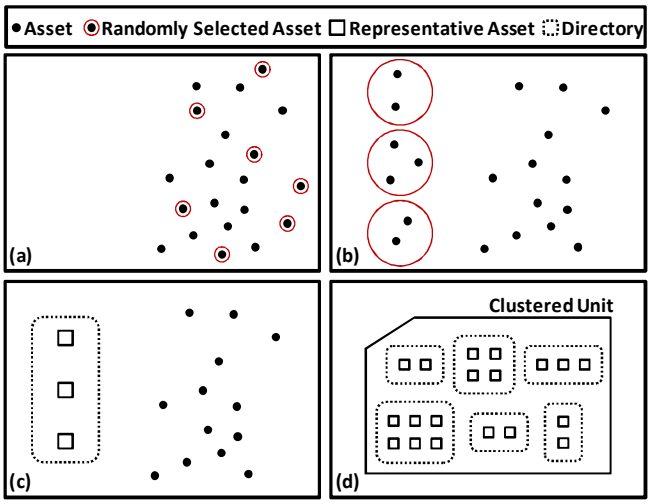

Figure 3. The first stage of the clustering algorithm in each storage unit.

To conclude the description of the first stage, it remains to explain how representative assets are generated. Note that the generation of representative assets is different for each type of asset. A representative asset, resulted from the combination of two clusters composed by dependent component specifications, includes all provided dependent interface specifications of the joined assets and the independent component specification they refer. This specification is the one that mostly occurs in the assets of the combined cluster. In turn, for clusters composed by independent component specifications, the representative asset includes all provided independent interface specifications of the joined assets.

A representative asset, resulted from the combination of two clusters composed by dependent interface specifications, includes all operations of the joined assets, as well as the independent interface they refer. This interface is the one mostly referred by the joined clusters. Taking into account a representative asset, resulted from the combination of two clusters composed by independent interface specifications, it includes all provided operations of the joined assets.

Finally, a representative asset, resulted from the combination of two clusters of component implementations, includes its referenced dependent component specification, which is also the one mostly frequent in the joined assets.

After generating all representative assets, the second stage takes place and joins together all clustered units, generating the whole clustered repository based on a pair-wised clustering scan. To do that, first, all clustered units are transferred to a single node that constitutes the distributed repository, in which the second stage takes place. Since clustered units are composed of representative assets only, the transmission of them to the selected node is faster than the transmission of the original storage units.

Thus, the second stage takes as input a set of directories, which have direct correlation with directories that belong to all clustered units. Once clustered units are stored in the selected node, the second stage adopts a multi-threaded, pair-wised clustering scan, in which pairs of directories are recursively processed and their representative assets are combined by applying the hierarchical clustering algorithm. At the end of the second stage, only one directory remains.

In order to join representative assets from directories A and B, the following procedure is applied. For each representative asset $a_i \in A$ and $b_j \in B$, the distance between them is computed, say $d_f(a_i, b_j)$. Let (a_k, b_l) be the pair for which the distance is minimum. If $d_f(a_k, b_l)$ is less than the defined threshold, then a_k and b_l are joined and the combined asset is added to B (also, a_k and b_l are removed from A and B); otherwise a_k is moved from A to B. The combinations of directories are performed in parallel, according to a divide and conquer strategy. Thus, pairs of directories are combined independently and successively until only one remains.

V. Results and Discussion

In order to evaluate the proposed approach, a set of experiments has been carried out for identifying the gains in terms of number of assets and storage space requirements between the clustered repository and the original repository. In experiments, the gains were evaluated taking into account a repository composed by 14000 X-ARM assets of different types, created by a customizable script that automatically generates them. After creating the repository, the proposed approach has been applied for grouping the stored assets in clusters, generating their respective representative assets.

Table I presents the number of each type of asset in the original repository and the clustered repositories after the application of the proposed approach using different thresholds, which vary from 100 to 200 in steps of 25. Note that the proposed approach significantly reduces the number of assets. As expected, the number of representative assets decreases as the threshold increases. When the threshold is increased, two assets have more chance of being considered

 ISBN: 978-1-61208-185-4

similar, and so, more chance of being grouped together. Thus, when the threshold increases from 100 to 200, the number of original assets reduces to 11176 and 7204 representative assets, respectively.

TABLE I. NUMBER OF ASSETS FOR DIFFERENT THRESHOLDS.

	Indep Int Spec	Dep Int Spec	Indep Comp Spec	Dep Comp Spec	Comp Impl	Total
Original	5000	3200	3200	1800	800	14000
Cluster 100	4104	2070	3023	1367	612	11176
Cluster 125	4053	2043	3012	1363	609	11080
Cluster 150	3386	1830	2634	1278	598	9726
Cluster 175	3315	1809	2628	1272	594	9618
Cluster 200	2168	1346	2045	1085	560	7204

For each considered threshold, the gain in number of assets has been identified and evaluated. Fig. 4 illustrates the gain in terms of the number of assets. For example, when the threshold is 150, the number of stored assets in the original repository is reduced around 30.5%, dropping from 14000 original assets to 9726 representative assets. As can be noticed in Fig. 4, the proposed approach performs a significant reduction in the number of stored assets, achieving relevant gains between 48.5% and 20.2%.

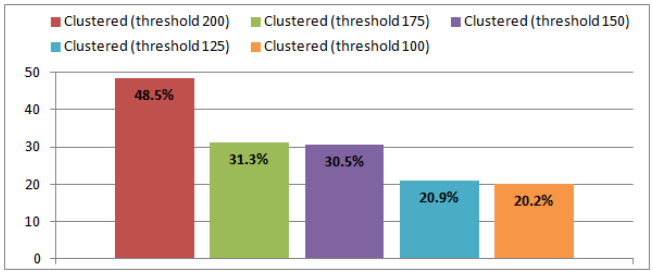

Figure 4. Total gain in number of assets.

However, as shown in Fig. 5, the gains are different for each type of asset. In general, the better gains are achieved for independent and dependent interfaces. For independent interfaces, the gains are between 56.6% and 17.9%. Considering dependent interfaces, the gains become a little bit more expressive, varying between 57.9% and 35.3%. Such higher gains can be explained by the considerable amount of assets of those types. As can be seen in Table I, the original repository has 5000 and 3200 independent and dependent interfaces, respectively. Thus, independent interfaces are the prevalent type of asset in the repository, increasing the likelihood of identifying similar assets.

In the case of dependent interface specifications, the gains become better due to three reasons. First, the number of assets of that type is also relevant. Second, in software projects, it is not rare to implement different versions of software systems for different target platforms. So, in component-based software projects, different versions imply on several dependent interface specifications for each independent interface specification. Considering that dependent interface specifications are considered similar when they refer to the same independent interface specification, it is easy to see that multiple implementations impacts on the likelihood of identifying similar dependent interface specifications. The third reason is a consequence of high gains in independent interfaces, which directly impact in the similarity metric of dependent interfaces.

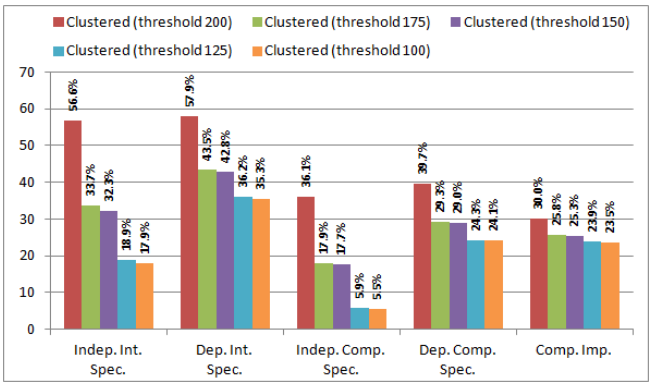

Figure 5. Gains in number of assets for different types of assets.

In relation to independent component specifications, despite the relevant number of assets (3200 assets according to Table I), the gains are notably low, varying from 36.1% to 5.5%. Besides, as can be noticed in Fig. 5, the gain of 36.1% occurs for the higher threshold only. When the threshold is 175 and 125, the respective gains decrease to 17.9% and 5.9%. Such gains are relatively low and indeed not expected. As mentioned before, independent component specifications are similar when they have in common a considerable subset of provided independent interfaces. Thus, it can be inferred that such low gains are a consequence of the difficulty of finding two or more independent component specifications that share a reasonable subset of independent interfaces.

In terms of dependent component specifications, surprisingly, the gains become more expressive, increasing to the range between 39.7% and 24.1%. It is possible to mention two reasons for such a surprising gain and both reasons are similar to effects pointed out before for dependent interface specifications. First, the existence of different versions of software systems for different target platforms implies on several dependent component specifications for each independent component specification. Considering that dependent component specifications are considered similar when they refer to the same independent component specification, it is clear to notice that multiple implementations impacts on the likelihood of identifying similar dependent component specifications. The second reason is a consequence of the gains in independent component specifications, which directly impact in the similarity metric of dependent component specifications.

Finally, considering component implementations, the gains are once more surprisingly good, varying between 30.0% and 23.5%. Taking into account that component implementations are considered similar when they refer to the same dependent component implementation, it is also possible to correlate such a good gain with the existence of different implementations of the same component specification, not only for different target platforms but also for meeting a variety of non-functional requirements, like performance, security and cost. Therefore, considering the various methods, techniques and algorithms that can be

 ISBN: 978-1-61208-185-4

employed to meet non-functional requirements, it is obvious that such multiple implementations impact on the likelihood of identifying similar component implementations.

Despite the mentioned gains, it is not enough to be efficient in reducing the number of assets, but also in downgrading storage space requirements for index files. So, after generating the clustered repositories, they were indexed using the indexing technique proposed in [7]. Table II presents the storage space required by all repository versions, after indexing them. As can be noticed, the proposed approach significantly reduces the required storage space, and, as expected, it decreases as the threshold increases. Thus, when the threshold increases from 100 to 200, the required storage space reduces from 12.50 to 9.22 MB.

TABLE II. STORAGE REQUIREMENTS FOR DIFFERENT THRESHOLDS.

Original	Clust 100	Clust 125	Clust 150	Clust 175	Clust 200
20.00 MB	12.50 MB	12.42 MB	11.35 MB	11.22 MB	9.22 MB

For each threshold, Fig. 6 illustrates the gains in terms of storage space requirements. For example, when the threshold is 150, the storage space required by index files reduces around 45.2%, dropping from 20.70 to 11.35 MB. Note that, the proposed approach significantly reduces storage space requirements, achieving gains between 55.5% and 39.6%.

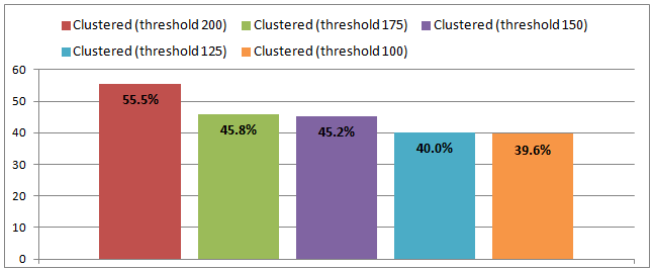

Figure 6. Total gain in storage requirements.

VI. CONCLUSION AND FUTURE WORK

Based on evaluation outcomes, it can be clearly observed the potential of the proposed approach in significantly clustering a large-scale, distributed repository and consequently reducing storage space required by index files. It must be highlighted that, the bigger the original repository in terms of the number of stored assets, the more expressive the likelihood of clustering assets, and so the better the gain in terms of storage space requirements.

Taking into account that the indexing technique proposed in [7] has an excellent performance in query processing time, even in large-scale index files, it is expected a reasonable gain in terms of query processing time due to the expressive reduction in the size of index files. Thus, the proposed approach clearly reduces storage space requirements, without introducing query processing time costs.

However, such expressive gains in terms of storage space requirements, almost certainly have an impact on the query precision level, since the process of clustering assets introduces some degree of information loss in representative assets. It must be stressed that the tradeoff between the best threshold and the query precision level has been initiated and preliminary results are encouraging, showing high values of precision and recall, which are two well-known metrics adopted in evaluation of information retrieval systems. Despite that, the impact of the proposed approach in terms of information loss still needs to be evaluated in more details. Besides, it is also under investigation a comparative analysis contrasting the distributed approach proposed herein and the centralized approach proposed in [8], both in terms of storage requirements gains and time complexity.

ACKNOWLEDGMENT

This work was supported by the National Institute of Science and Technology for Software Engineering (INES – www.ines.org.br), funded by CNPq, grants 573964/2008-4.

REFERENCES

[1] A. Orso, M. J. Harrold, and D. S. Rosenblum, “Component Metadata for Software Engineering Tasks”, Proc. 2nd Int. Workshop on Engineering Distributed Objects, 2000, pp. 126-140.

[2] P. Vitharana, F. Zahedi, and H. Jain, "Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis", IEEE Trans. on Soft. Engineering., vol. 29, issue 7, July 2003, pp. 649-664.

[3] OMG, Reusable Asset Specification – Version 2.2, 2005.

[4] G. Elias, M. Schuenck, Y. Negócio, J. Dias, and S. Miranda, “X-ARM: An Asset Representation Model for Component Repository”, 21st ACM Symposium on Applied Computing, 2006, pp. 1690-1694.

[5] W. Meier, “eXist: An Open Source Native XML Database”, NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems, 2002.

[6] R. Goldman and J. Widom, “DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases”, 23rd Int. Conf. on Very Large Data Bases, 1997, pp. 436-445.

[7] T. Brito, T. Ribeiro, and G. Elias, “Indexing Semi-Structured Data for Efficient Handling of Branching Path Expressions”, 2nd Int. Conf. on Advances in Databases, Knowledge, and Data Applications, 2010, pp. 197-203.

[8] M. P. Paixão, L. Silva, T. Brito, and G. Elias, “Large Software Component Repositories into Small Index Files”, 3rd Int. Conf. on Advances in Databases, Knowledge, and Data Applications, 2011, pp. 122-127.

[9] R. C. Seacord, “Software Engineering Component Repositories”, Technical Report, Software Engineering Institute (SEI), 1999.

[10] W. Frakes and K. Kang. “Software Reuse Research: Status and Future”, IEEE Trans. on Soft. Engineering, vol. 31, no. 7, July, 2005.

[11] J. P. Oliveira, T. Brito, A.E. Oliverira, S. E. Rabelo, and G. Elias, “X-CORE: A Shared, Disributed Component Repository Service”, 24th Tools Session / 21st Brazilian Symp. on Soft. Engineering, 2007, pp.100-106 (in portuguese).

[12] C. Boldyreff, D. Nutter, and S. Rank, (2002) “Open-Source Artifact Management”. Workshop Open Source Sof. Engineering, USA, 2002.

[13] R. Xu and D. Wunsch, “Survey of Clustering Algorithms”, IEEE Trans. on Networks, vol.16, issue 3, pp. 645-678, May 2005.

[14] A. K. Jain and R. C. Dubes, Algorithms for Clustering Data, Prentice Hall, 1984.

[15] S. Mancoridis, et al. “Using Automatic Clustering to Produce High Level System Organizations of Source Code”. Proc. IEEE Int. Workshop on Program Comprehension, pp. 45–53, 1998.

[16] B. S. Mitchel and S. Mancoridis. “On the Automatic Modularization of Software Systems Using the Bunch Tool”, IEEE Trans. on Soft. Engineering, vol. 32, issue 3, pp. 1-16, March 2006.

[17] Y. Chiricota, F. Jourdan, and G. Melançon, “Software Component Capture using Graph Clustering”, IEEE Int. Workshop on Program Comprehension, 2003.

[18] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of Clustering Algorithms in the Context of Software Evolution", 21st Int. Conf. on Soft. Maintenance, 2005, pp. 525-535.

[19] Z. Li, Q. A. Rahman, and N. H. Madhavji, “An Approach to Requirements Encapsulation with Clustering”, 10th Workshop on Requirement Engineering, 2007, pp. 92-96.

 ISBN: 978-1-61208-185-4