Web Service Clustering using Multidimensional Angles as Proximity Measures

CHRISTIAN PLATZER, FLORIAN ROSENBERG, and SCHAHRAM DUSTDAR
Vienna University of Technology

Increasingly, application developers seek the ability to search for existing Web services within large Internet-based repositories. The goal is to retrieve services that match the user's requirements. With the growing number of services in the repositories and the challenges of quickly finding the right ones, the need for clustering related services becomes evident to enhance search engine results with a list of similar services for each hit. In this article, a statistical clustering approach is presented that enhances an existing distributed vector space search engine for Web services with the possibility of dynamically calculating clusters of similar services for each hit in the list found by the search engine. The focus is laid on a very efficient and scalable clustering implementation that can handle very large service repositories. The evaluation with a large service repository demonstrates the feasibility and performance of the approach.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Measurement, Performance

Additional Key Words and Phrases: Web service, search, vector space, discovery, clustering, service discovery, search engines

ACM Reference Format:

Platzer, C., Rosenberg, F., and Dustdar, S. 2009. Web service clustering using multidimensional angles as proximity measures. ACM Trans. Internet Technol. 9, 3, Article 11 (July 2009), 26 pages. DOI = 10.1145/1552291.1552294 http://doi.acm.org/10.1145/1552291.1552294

1. INTRODUCTION

With the growing popularity of Web services throughout the service-oriented community [Papazoglou 2003; Papazoglou et al. 2006], the methods for service

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).
Authors' addresses: C. Platzer, Distributed Systems Group, Information Systems Institute, Argentinierstrasse 8/184-1, 1040 Vienna, Austria; email: cplatzer@seclab.tuwien.ac.at; {florian, dustdar}@infosys.tuwien.ac.at.

DOI 10.1145/1552291.1552294 http://doi.acm.org/10.1145/1552291.1552294

discovery and search in repositories grow more sophisticated. The establishment of common description standards for Web services, such as WSDL [Christensen et al. 2001], was an enabler for different research efforts in this area, especially those related to service registries and their corresponding retrieval mechanisms. Particularly in registry-like directories, a huge number of services have to be indexed and retrieved in a fast and efficient way to ensure scalability. An additional need that arises with the growing number of entries is to automatically create a list of related services that match a given query. Such rudimentary search capabilities are more or less part of every Web service registry. In most cases though, these registries do not focus on the search functionality or its efficiency. UDDI registries [OASIS 2005] retrieve their contained data by searching tModel entries for a match to a given inquiry request. Some forms of registries use a simple full text search to react to queries while others try to produce matches based on some form on semantic index.

Apart from the direct relation to the issued query, there is currently no established method to relate these possible matches to each other. The goal is to enrich a search engine's hits with a number of related services that fulfill a similar task. Such a functionality would be an enormous help in browsing the content of registry-like service directories. It is important to understand though, that the original search result is unaffected by those relations. The clusters are built for each element of the search result and aim to provide a set of possible alternatives for each entry in a result list.

In this article, an approach is presented that uses statistical cluster analysis to create the desired containment for the highest matches of a given query. The focus is laid on an efficient algorithm that is scalable to large and distributed service repositories and still guarantees an efficient processing of queries and subsequent clustering for the generated matches. For this purpose, the usual Euclidean distance for proximity measurement is extended by means of a multidimensional angle produced by a vector space search engine. Furthermore, the very complex runtime creation of the matrix for the distance measurements is discussed and changed to a more effective method. This change is necessary, because large service repositories otherwise result in cubic runtime complexity and are therefore limited in their processing capacities.

The rest of this article is structured as follows: In Section 2, we present the necessary prerequisites to perform cluster analysis on the given data structures. The topics discussed include quantification as well as normalization. Section 3 presents the possible ways to perform service clustering for services with, and services without, domain-specific information. Furthermore we evaluate the most promising algorithms for our approach. In Section 4, we present the implementation of the aforementioned clustering concepts within our vector space search engine followed by an evaluation of the work in Section 5 In Section 6, we present an overview of important related work. The approaches discussed do not always deal with service clustering explicitly but often take a larger view, reaching into semantics as well as metadata generation for Web services. Finally, the last section concludes the article and presents some of our future work.

2. PREREQUISITES

This section gives an overview of the most important issues related to general clustering problems and Web service clustering in particular. The principal method is a modified version of common statistical cluster analysis [Eckey et al. 2002]. Because some of the statistical methods are not directly applicable here, some adjustments have to be made to ensure they can still be used.

2.1 Requirements

Statistical cluster analysis can be used for a broad spectrum of input data, ranging from Boolean values or even nominal scales, to relational scales. The more unambiguous the data for the different variables can be set, the more significant the clustering result will be. Therefore, it has to deal with the requirement to have float or integer values for a numerical representation of a service description. Furthermore, the cluster algorithm must not be limited to a specific number of variables and/or a maximum size for the stored entries to allow it to be executed on a multidimensional term space. With these requirements in mind, the indexing method can be examined.

2.2 WSDL Indexing

When processing a WSDL description in general, the desired result is an index where each characteristic is represented as a dimension of an n-dimensional vector space. The meaning of the dimensions strongly depends on the indexing procedure used. If domain-specific information is characterized by a dimension, for example, the cluster analysis will produce a result where elements of similar domains are grouped and identified as related to each other. How well the desired outcome matches the expectations of a query therefore depends on the quality of the index as well as the algorithms used. The following types of index structures are possible when dealing with XML-based service descriptions.

2.2.1 *Syntactic Indexes.* A syntactic index has the advantage of being able to process any valid WSDL file from any source simply because input data is not restricted by any means. In Platzer and Dustdar [2005] such an index is used to create vectors for WSDL files and process queries upon them by calculating document similarities. The basic idea is to parse through WSDL documents and extract as many keywords as possible. This method is a common approach used in all fields of natural language processing. The list of keywords is then mapped to a vector space, where every dimension represents a characteristic or in this case a keyword. All dimensions together then span an n-dimensional vector space, where n is the number of characteristics that have been quantified.

Using these indexing methods, the following elements are extracted:

—*Types*. The type names and their corresponding attributes are extracted. Those types are usually assigned by developers and give clues about the functionality; for example, “CardNumber” or “customerId.” The words are split up and indexed accordingly.

—*Messages*. Similar to types, messages allow a specific functionality and are created by programmers in most cases. Sample message names are “doGoogleSearch” or “getStockQuote.” Just like types, they can be split and indexed.

—*Port types*. Port types finally define operations, where certain messages are composed as input and output. Here, the names of the operations are interesting and are considered the same way, message names are.

—*URLs*. The endpoint URLs of a Web service description holds valuable information about the domain: where it is registered, the implementation, and any other important information such as service location and service owner. Properly processed, these values can help to build service clusters.

—*Comments*. Comments are among the most important information a WSDL description contains when it comes to searching. Most of the comments are written by humans or sometimes generated by service engines. For the clustering approach, comments are not of the topmost priority, because they are not expected to hold a large amount of domain-specific information. In reality, commentary sections are used quite rarely. From a variety of 250 distinct and functional real-world Web service descriptions, just three contained comments that were entered by a human. The rest either contained generated comments (most frequently by the servlet engine) or no comments at all. Therefore, comments are handled with low priority and a low importance rating when it comes to clustering.

Although there are other elements of a WSDL file, they are not considered here because of their limited information. The service’s *binding* is a good example. It just assembles all available port types and assigns a transport means, which is not interesting for the keyword generation process.

2.2.2 *Rich Indexes.*

As contrasted with purely syntactic indexes, a rich index deals with some sort of specialized information contained in the original input data. The information can be of various structures. The four most important values are listed here.

—*Semantic descriptions*. When a WSDL description is enriched with some sort of semantic descriptions like RDF[W3C 2000], this data can be parsed and stored for further use. The information can be mapped to a domain-specific ontology in most cases and processed accordingly. For this purpose however, this is not an option because no real-world service really implements semantic descriptions in the form of RDF tags. The same applies for semantic annotated Web service descriptions (SAWSDL).

—*Domain information*. Other than semantic descriptions that might be entered directly into the WSDL description, it is possible to query a user for domain information about a specific service. Unfortunately it cannot be assumed that every service comes with such information attached. If it is the case nevertheless, it can of course be used to build a rich index.

—*Location information*. It can be gathered from the endpoint of a given service. There are services like GeoIP[1] that make it possible to determine the location and additional information using the endpoint IP of a service.

—*QoS descriptions*. Similar to domain and location knowledge, QoS descriptions for performance related aspects of Web services are a type of meta-data that can be gathered for Web services[Rosenberg et al. 2006]. When the evaluated QoS is used to build indexes, the clustering algorithm will produce groups of similar performance values. This can be an intended behavior or an unwanted side-effect depending on the desired clusters that should be produced as a result.

Although it is possible to merge information from these categories with syntactical information, it is not recommended. The information within a rich index is usually already structured. Therefore, vectorizing these elements and applying information-retrieval techniques is not necessary. Semantic descriptions for instance, can simply be queried to produce exact matches. Furthermore, vector spaces built from service descriptions tend to span a large number of dimensions. Even when additional information like domain information produces an exact match, the multitude of other vectors that have to be considered is likely to move such vectors apart from each other, and therefore, reduce the effectiveness of the clustering algorithm.

Finally, an index that is based on QoS will not benefit from a combination with natural language processing, even when the numerical structure is identical. A search issued on a combined space would result in a broken search semantic. An example will clarify this situation. A cluster analysis based on a search string like “credit card verification” expects to find clusters of the same domain. With a merged QoS index however, the result would possibly contain a service that verifies lotto numbers but has the same QoS attributes and list it as very strongly related. Therefore, vector spaces should not be merged across different index strategies.

In this particular case the most feasible and also most depictive indexing method is syntactic clustering. First, the data cloud produced is of the highest density and on the other hand it forces the changes that allow the algorithm to operate on spaces with unbounded dimensionality.

3. BASIC CONCEPTS OF STATISTICAL CLUSTERING

Assuming there is an already established vector space for an existing WSDL repository, the clustering approach can be discussed in detail. As mentioned in the introduction, traditional statistical cluster analysis does apply in a Web service environment with certain limitations. Those limitations and how to overcome them are discussed in this section.

[1]http://www.maxmind.com/app/ip-location

3.1 Proximity Measure

When dealing with metric scale levels as in this approach, some ways exist to measure the distance between two elements. Two of these norms are most commonly used.

3.1.1 *City-Block Distance.* In an m-dimensional vector space, the City-Block distance between two elements j and k is calculated with:

$$d_{jk} = \sum_{i=1}^{m} |x_{ij} - x_{ik}|.$$

This distance measurement is designed to be used for a data cloud where elements are not very different from each other. The absolute value for the difference on each axis is added pairwise. As a result, one dimension with a large deviance results in a large distance for the tuple as a whole.

3.1.2 *Euclidean Distance.* Although the City-Block distance is a valid measurement for this purpose, the Euclidean distance has the advantage of considering the direct distance between two points in the vector space, no matter how large the value for a particular dimension is. The Euclidean distance is calculated as follows:

$$d_{jk} = \sqrt{\sum_{i=1}^{m} (x_{ij} - x_{ik})^2}.$$

A variation of this measurement is the squared Euclidean distance. Its only difference from the standard method lies in omitting the square root for the final value.

Both of these methods are theoretically applicable in this environment. Nevertheless, the practical use shows the limitations and restrictions. The first step when splitting a data cloud into clusters is to calculate distances for every pair of elements in the space. We use the term *element* for every dot in the space, because they will be used to represent an element of our service repository (a WSDL file) later in the article. Consequently, the term *dimension* refers to one dimension of the matrix on which the clustering should be performed. For statistical use, the number of elements is usually limited to amounts of around 100. The number of dimensions rarely exceeds 100 as well. For that amount of data, the values can be processed relatively quickly and precisely. For WSDL repositories however, 10^4 entries are considered medium size. Even though the amount of currently available entries is far less, the observation explained in Section 1 implies a growth in service repository size. Furthermore, the dimensional size of the vector space is also considerably larger. Depending on the input data, 10^2 to 10^3 dimensions is a reasonable amount for a single WSDL document and 10^4 for the whole vector space. With those assumptions in mind, the expense of processing standard distances can be calculated:

(1) All distances for all elements within the vector space have to be processed. Permuting n elements without repeating, results in a Gaussian progression

for the number of needed iterations:

$$\#Iterations = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

So the number of iterations alone results in an upper bound of $\mathcal{O}(n^2)$ for the problem complexity. Additionally, each iteration needs to process as many elements as there are dimensions present. It is possible to precompute the distances on a local repository. With a distributed vector space, however, access to all vectors is not guaranteed, which impedes a synchronous processing of the vectors.

(2) In the next step, a cluster algorithm has to be applied, where near elements are grouped by either hierarchical or agglomerative procedures. Assuming the worst case scenario, each application of the cluster algorithm results in a pair of two elements, leaving $n - 1$ elements for the next iteration of the algorithm. As a result, the cluster algorithm consumes an additional $(n - 1) \Rightarrow \mathcal{O}(n)$ iterations in the worst case.

(3) Finally these results, and possibly the results of remote vector spaces can be combined and displayed.

Such a high computation expense for the cluster analysis is usually compensated by precomputing these values and updating them when the repository changes. Unfortunately, this method is only applicable in a centralized approach and not in a distributed environment as the one at hand. Besides these performance-related drawbacks, there is also a problem where documents of different length are not considered in one cluster because they are represented by a different cardinality. As an example, two descriptions are taken that are both of the financial sector. After the indexing phase, both documents are represented by the keywords "interest" and "investment." Because one document is longer than the other and the keywords occur more often, the cardinality is different, which means that the Euclidean distance puts those two documents in different clusters even though they are strongly related.

With all these restrictions it is obvious that a better way to compute distance values has to be found; one that copes with performance and precision at the same time.

3.1.3 *Multidimensional Angle.* An elegant solution for similarity or distance ratings in an n-dimensional vector space is to calculate multidimensional angles between the elements. In this approach, it is not the absolute position of two points (p, q) in space and the Euclidean distance between them, but the cosine of the angle between two vectors reaching from the origin to (p, q). It is calculated by the following formula:

$$cos\,(p,q) = \frac{p \cdot q}{\|p\| \cdot \|q\|} = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2 \sum_{i=1}^{n} q_i^2}},$$

where n is the number of overall dimensions.

One of the advantages of this method is that the vectors are already normalized. Therefore, a document's length does not influence the distance measure.

In the previous example the vectors produce the same angle and are therefore considered to be close to each other.

In terms of performance, this approach also produces better results. When taking a sample vector space with 10^4 dimensions, it can be assumed that a single document does not incorporate all different dimensions. Therefore, dimensional reduction can be applied while computing angles for two vectors. In this particular case, a dimension is only considered when it is present in both vectors. Otherwise, it would drop out of the equation anyway. The results show the saved computing time for a single vector query.

This approach also provides the possibility of producing every angle for a single dimension in the same iteration by storing denominators and reusing them for other elements as shown in Platzer and Dustdar [2005]. With this method, the expense of producing the distance matrix can be reduced to $\mathcal{O}(n)$, which is an acceptable but still improvable growth rate. For high-performance search engines where results are created on the fly, this leaves just two options.

—All distances are processed in advance and recalculated as soon as a new entry is added to the vector space. This results in a huge amount of processing time for adding elements. Furthermore it strongly affects the distribution capabilities of the whole approach.

—Clusters are built for results of a search query only. That limits the amount of elements to a reasonable size and also improves visibility of the result.

In our implementation the latter solution is preferable because of its distribution capabilities. With the limited size of the repository, the runtime overhead is tolerable.

3.2 Cluster Algorithm

For the final cluster algorithm, quite a large range of possibilities exists but it is out of scope of this article to discuss all of the advantages and disadvantages.

In general, partitioning and hierarchical methods can be distinguished. Depending on the underlying data structure, various different algorithms to create clusters in data clouds exist. The k-means method is among the most popular of them [la Torre and Kanade 2006] and therefore was our first choice for our application. Structurally, k-means is a partitional cluster method. The algorithm assigns each point in the data cloud to the cluster whose center is nearest. The center is simply the average of all points in the cluster. For multiple dimensions that means that each coordinate is the arithmetic mean of this dimension for all points belonging to this cluster. The original k-means algorithm as proposed by MacQueen [1967] consists of the following steps.

(1) Choose the number of clusters k.
(2) Randomly generate k clusters and determine the cluster centers, or directly generate k random points as cluster centers.
(3) Assign each point in the data cloud to the nearest cluster center.
(4) Shift the new cluster centers according to the added points.
(5) Repeat Steps 3 and 4 until some convergence criterion is met.

With some minor adoptations and performance optimizations, this algorithm is widely used to generate clusters for all different kinds of quantified data. For our purpose there are two drawbacks. First, the number of clusters has to be predefined. Unfortunately there is no way to estimate how many items of the original search result are strongly related to each other. Therefore a rule of thumb—like the elbow criterion for example—would have to be applied to appraise the number of clusters that make sense in the final result. The second drawback of this method is that it does not yield the same result with each run, since the resulting clusters depend on the initial random assignments. For these reasons we decided to use a very similar but hierarchical method with an agglomerative approach for our problem. Unlike the k-means method, each element is considered as a single cluster at startup. With each iteration, new clusters are built, and contained elements are grouped to the new layout before the algorithm is restarted. Just like the k-means algorithm, all elements are finally distributed to a cluster with the only difference being that the number is denoted by the iteration step and not defined at the start. For a better visualization however, a hierarchical method with a centroid fusion algorithm is preferable to the partitional approach. It performs well for the fusion process and is limited in the necessary iteration steps. The algorithm is applied as follows.

(1) Search for the pair in the distance matrix with the minimum distance $d_{min}(a, b)$.
(2) Create a new distance matrix where distances between clusters are calculated by their mean value $d(\bar{a}, \bar{b})$.
(3) Save the distances and cluster partitions for later visualization.
(4) Proceed with Step 1 until the matrix is of size $n = 1$, which means that only one cluster remains.

To give a better understanding of the algorithm involved, we provide an example with a matrix of size 5, including all necessary steps for the matrix reduction. The algorithm starts to build the initial matrix by querying all relations to item A. If a relation exists, it is entered into the matrix with its corresponding value, and zero otherwise. A sample initial matrix is shown in Table I(a). Issuing a query with the vector of item A for example would result in a rating of 0.8 for item E, 0.55 for item D, and so forth. Each element is processed this way until the necessary elements are entered into the compressed matrix. Then the matrix is decompressed, to ease the following iteration steps. To do so, the main diagonal has to be filled with the element that represents the strongest relation (in this case 1). All other values can simply be mirrored due to the bijective nature of the document relations.

The algorithm is processed in the specified order. The highlighted element is the minimum distance in the current reduction step, and therefore determines which elements will be combined for the next step. Higher values mean a smaller distance or put differently, higher similarity, because they represent the cosine of the angle between two vectors. The values on the main diagonal are not considered here. Furthermore, this value denotes the coefficient

Table I. Matrix Reduction Example

(a) Compressed initial Matrix

	A	*B*	*C*	*D*	*E*
A	—	0.3	0.5	0.55	0.8
B	—	—	0.7	0.6	0.85
C	—	—	—	0.9	0.4
D	—	—	—	-	0.1
E	—	—	—	—	—

(b) Decompressed initial Matrix

	A	*B*	*C*	*D*	*E*
A	1	0.3	0.5	0.55	0.8
B	0.3	1	0.7	0.6	0.85
C	0.5	0.7	1	**0.9**	0.4
D	0.55	0.6	**0.9**	1	0.1
E	0.8	0.85	0.4	0.1	1

(c) Reduction step 1 (4 Elements)

	A	*B*	$\overline{CD}$	*E*
A	1	0.3	0.525	0.8
B	0.3	1	0.65	**0.85**
$\overline{CD}$	0.525	0.65	1	0.25
E	0.8	**0.85**	0.25	1

(d) Reduction step 2 (3 Elements)

	A	$\overline{BE}$	$\overline{CD}$
A	1	**0.55**	0.525
$\overline{BE}$	**0.55**	1	0.45
$\overline{CD}$	0.525	0.45	1

(e) Reduction step 3(2 Elements)

	$\overline{ABE}$	$\overline{CD}$
$\overline{ABE}$	1	**0.4875**
$\overline{CD}$	**0.4875**	1

(f) Termination step

	$\overline{ABCDE}$
$\overline{ABCDE}$	**1**

that enables the visualization of the cluster. In each reduction step, the new matrix is shrunk by one element and the new matrix elements are calculated as an arithmetic mean value until the matrix reaches its trivial state of two remaining elements, as shown in Table I(e), where the last distance is shown and the matrix reaches the termination state I(f). The last remaining value, which is 0.4875 in this case, denotes the distance between the centers of the two remaining clusters before they are fused. An agglomerative algorithm always processes the whole data cloud until one single cluster remains. How the reduction steps are used can be decided later.

3.3 Results

In the result phase, the clusters can either be visualized in an elbow-diagram or as a dendrogram. The advantage with already normalized values is that they are always in a range $[0, 1]$. Additionally, the distance matrix gives a good idea of the different steps the algorithm went through. With all distances calculated this example, it is finally possible to visualize the cluster distances in such a dendrogram. Figure 1 shows the strong relation of the items *C,D* and *B,E*. Although the layout might suggest otherwise, the dendrogram is not to be mistaken for a hierarchical organization, because the cluster elements are equal. It merely describes the distances of the produced clusters.

With this example it also becomes clearer why it is so difficult to use predefined numbers of clusters or a preset termination target. With the clusters set to two for example, the result would just show the elements *C,D* and *A,B,E* as part of a cluster but not how strongly they are related to each other. Furthermore, when setting a termination distance of 0.75, the algorithm would end up with just one cluster which contains all elements. By looking at the dendrogram, however, it immediately becomes clear which elements are tightly grouped.

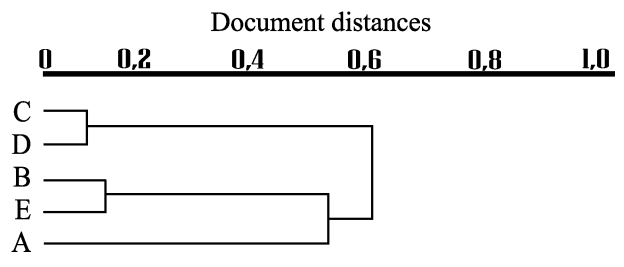

Fig. 1. Dendrogram visualization.

In the next section, we give an explanation of how to implement this rather theoretical approach in a Web-based search engine.

4. IMPLEMENTATION

To evaluate the efficiency and usability of the proposed approach, we embedded an implementation into our Web-based search engine. A discussion of the performance measures and scalability issues based upon that implementation forms the main element of this section.

We decided to implement our prototype with a Web-based interface that requires no additional features or installations to run it and test the underlying functionality. The Web page is publicly available and part of the already established WSDL search engine, V-USE.[2] The environment of this implementation is provided by the VitaLab[3] laboratory at our institute. This infrastructure is designed to allow deployment of a multitude of services on the lab machines, which enables an easy integration of research prototypes and the necessary applications to evaluate them. Supporting various kinds of frameworks like ASP.NET on IIS, Apache, Axis and the like, this environment gives the freedom to choose the best solution to implement a particular research prototype. Each of the integrated machines encompasses 2 Dual Core Intel Xeon CPU's with 3.2 GHz, 1GBit network interface, 2 GB of main memory and 10krpm RAID 1 HDs. The machine where V-USE is deployed runs on Microsoft Windows Server 2003 Enterprise Edition with Service Pack 1.

We utilized a decoupled development of the search engine and the application for the user interface. Because of this structure, it is possible to plug in the clustering functionality at two different points.

(1) The vector space itself is handled in the backend. The advantage of this location is simply a better debugging capability and improved performance. The performance gain is a result of the centralized structure of the search engine. Processing n queries for a n-Element Cluster means the same amount of Web service invocations as, if handled by the front-end application. Furthermore, the execution time for each single query, as well as the overall times, can easily be extracted in this way by simply logging them to the Java runtime environment.

[2]http://vuse.de.vu/.
[3]http://www.vitalab.tuwien.ac.at/

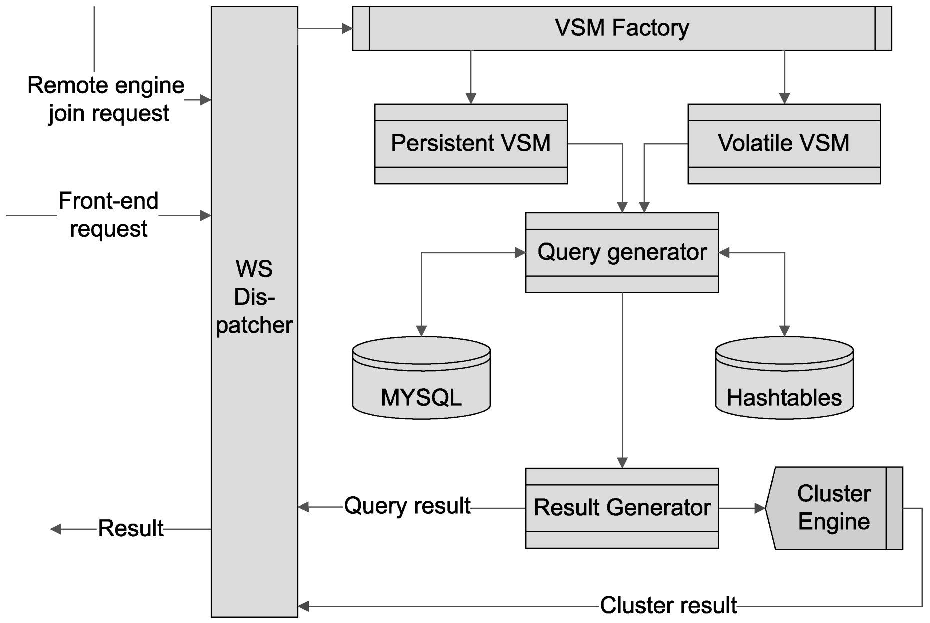

Fig. 2. System Architecture with cluster plugin.

(2) The frontend on the other hand has one big advantage. The original vector space engine is designed to allow repositories to be split into several smaller repositories while queries can still be executed upon them as if on one single repository. The already implemented method for runtime weighting and normalization is able to map invalid vectors from spaces with different dimensional layout from any vector space. The benefit gained is the ability to split repositories in several sub-repositories whenever the performance is not satisfactory, and still keep the dimensional structure and relations intact. For the clustering approach, it would therefore make sense to execute the queries to fill the cluster matrix at the client side where the distributed spaces are joined in the first place. Nevertheless, it was decided to implement the prototype with the method mentioned first, because distribution capabilities are not issues for development.

As a result, the clustering functionality is realized as an extension of the original Web service and can be seen as a plug-in for the vector space engine itself. The system layout, as depicted in Figure 2, shows the general architecture including the cluster plug-in as it is implemented in the back-end-based approach. With this structure, a completed request causes the following order of events.

(1) A request to the Web service endpoint reaches the axis servlet. Whether the request was sent by the front-end or another vector space implementation is not important at this point.
(2) Depending on the request signature, the vector space Factory decides whether to instantiate the persistent or volatile version of the search

engine. The volatile engine is blanked every time the tomcat servlet engine is restarted, because the vector matrix is implemented with hash tables. In the persistent version, this limitation does not apply naturally. Instead, the constructor makes sure that the requested database structure is valid. If not all tables exist, they are created on the fly, thus providing an empty vector space on startup.

(3) For a normal query, the query generator decides whether to directly access the local hash tables, or create the corresponding SQL statements to fetch the same data from the database. It is important to remember that those queries effectively implement a dimensional reduction of the original query vector. This means that even for vector spaces with a large dimensional count, the processed data can be kept low. Search queries normally do not exceed a size of 10 words, and therefore dimensions. Documents that do not contain any of the query words at all, are automatically omitted and, therefore do not cost any processing time. This advantage does not apply to the cluster engine because when building the matrix, every element of the original vector must be taken into account, which results in large query vectors of 100 and more dimensions. Furthermore, it is almost certain that every document keeps at least one of the vector elements, and therefore, will also produce a query rating. The actual numbers are discussed in the next subsection.

(4) After the query is executed, the result generator takes over and either produces a sorted list of results that can be handed to the Web service dispatcher, or fill the matrix of the cluster engine.

(5) The cluster engine decompresses the matrix, and reduces it step by step until the cluster algorithm is successfully finished.

(6) The Web service dispatcher can now deliver the results to the caller and free all used resources.

This scenario describes how the back-end reacts to a request and executes the corresponding operations. A more precise evaluation of the involved time frames and the delay caused by the cluster algorithm, can be created by utilizing the prototype implementation with the Web interface [Platzer 2007].

5. EVALUATION

An exact and significant evaluation of prototypes and implementations for Web service technology is always a very challenging undertaking. Partially because the usefulness of a new method is hard to prove in a prototype, but basically because of a lack of sufficient real-world services. A restricted number of descriptions for actually working services is not a big issue for fundamental implementations that deal with services at a functional level. For search engines and metadata related research though, this fact produces enormous problems. Repositories with around 1000 distinct services are often too small to pose a real problem for the involved algorithms. The only possibilities left are to either produce copies of already existing services, or create dummy services with no actual implementation to simulate larger repositories [Magdalenic et al. 2006].

In either case, the drawbacks are quite obvious: either duplicate entries and therefore insufficient search capabilities, or biased results because of the generation algorithms to produce the services. In most cases, the best method is to base an evaluation on a real repository and assess performance and scalability capabilities by extrapolation.

5.1 Prototype Execution

As mentioned earlier, the file base is a critical issue for the numerical evaluation of such an approach. The repository we use for testing purposes contains a set of 275 distinct WSDL files from different sources. Some of them were extracted by querying UDDI registries and a cross-reference download of contained keys. The big disadvantage with UDDI was that WSDL descriptions had no designated position but could be stored in various ways. This fact made an automated extraction quite difficult. Furthermore, the biggest UDDI registries from Microsoft [Microsoft 2005] and IBM [IBM 2005] were shut down some time ago. Therefore, UDDI is no longer an option for gathering public service descriptions.

The second source we used was the public Web service registry from xmethods.[4] Other than UDDI registries, xmethods provides a multitude of possibilities for accessing the underlying services, reaching from standard Web pages to RSS feeds and even a Web service interface to query the database. Additionally, the functionality for populating Web services with a set of input parameters is provided. This way, the services can be tested for their availability directly from the Web site.

With this repository, the vector space engine can be populated and readied for a cluster analysis. To produce demonstrable results, we chose two different queries for the initial search. One aims to find the description of the popular Amazon Web service. The corresponding query string is "Amazon web service". The other tries to find a service for verifying credit card numbers. Here the query is "Credit card verification."

Both of the initial searches took 16 milliseconds to finish upon the 275-Element repository, using the persistent vector space engine. Looking at the queries one will see that both of them encompass three dimensions. Starting from this point, the files "wsCreditVerify.wsdl" and "AmazonWebService.wsdl" were the best and most desired results found by the search engine. After completing the search, the clustering functionality is available for each result by clicking the "cluster" button for the selected result element. Completing this initial search as quickly as possible is an important matter for the search engine. Otherwise the Web page would respond slowly, which is a very limiting fact for such a facility. The most important steps when processing the query are as follows.

(1) The query word is taken and normalized by the same algorithm that filters the WSDL files. With the first string, the result is "amazon web service." This normalization filters spaces, eliminates alphanumeric signs or cuts spaces where they are not needed.

[4]http://www.xmethods.org/

(2) Now the result can be put into a vector. To do so, a weight has to be applied for each element. Balanced search is achieved with a weight of 1 for each dimension. So the final vector is (1,1,1) with (amazon,web,service) as the corresponding terms. It is also possible to weight terms otherwise, based on linguistic resources for example.
(3) In the next step, all remote repositories are queried for their statistics with the given vector as input. This is technically like an ordinary query with the difference that results don't have to be sorted, which saves computing time. Also it can happen simultaneously on each remote host.
(4) All statistics are merged and finally each vector space can be queried with the vector and the accumulated statistics. The results are again merged according to their relevance and displayed afterwards. When processing local vectors only, Step 3 can be omitted. This example produces a relevance rating of 0.953 for the amazon query on the local repository.

To proceed with the clustering, the next step in the execution chain is to recreate the original vector of the initial root Element. The Amazon Web service will serve as an example. Here, the WSDL description file responds with 210 dimensions/keywords in the vector space. The high dimensionality means the query that has to be executed on the vector space for the matrix creation will cause a significantly larger load than a normal search query.

The settings of the cluster engine are such, that the initial matrix is of size 15×15. This value can freely be chosen in the Web application. It defines, how many results of the query with the root element are taken into the matrix. It's important to remember, that the queries are executed the same way an ordinary query gets processed. Therefore, the matrix is filled with elements from the whole repository. It also means, the same number of queries has to be executed to fill the matrix. The number was chosen because it provides a fair tradeoff between good performance and a meaningful result. The 15 elements that build the matrix of the Amazon-Cluster took an average of 165.5 ms, plus an additional 65 ms for the sorting algorithm of the result. Compared to the 16 ms of the initial query, the impact of the dimensional reduction algorithm becomes clear. That means, a matrix of size 15 takes about 3.5 seconds to fill. This time does not directly depend on the size of the underlying repository but on the speed of the executed queries. Therefore, with a growing size of the desired elements comprised by the final cluster, the time to fill the matrix grows linearly, independent of the repository size.

After the cluster matrix is filled, the cluster algorithm itself can proceed in its execution and reduce the initial matrix step by step until it is of size 1. The matrix reduction for a 15-element matrix takes a constant time and finishes in approximately 150 ms on the test machine. The result of the overall procedure is a list of n elements and the clusters they belong to. The processing steps are exactly as described in Section 3.

(1) The vector resulting from the initial search is used as a normal query. The best 15 results are the elements of the compressed matrix. Alternatively, it is also possible to use the original result of the query and fill the matrix

Table II. Document Names

Document Number	Service Name
0	AmazonWebServices
1	ECOWS WS sample
2	GoogleSearch
3	GoogleSearch(2)
4	google_search_service
5	GoogleSearch(1)
6	Exchanges
7	xExchanges
8	ElmarSearchServices
9	ws4lsql
10	WolframSearch2
11	InsiderTransactionInfo
12	JetFoldersService
13	xHoldings
14	ZacksCompany

with these elements. Doing so would result in a matrix that encompasses a thinned space population, because only elements of the result are considered. For the time being, the formally complete method with a whole vector is used. The other possibility is discussed later.

(2) A query with the vector of each element in the matrix must be processed to fill the matrix for all other elements.

(3) The result is decompressed to a 15×15 matrix. To do so, the main diagonal is filled with 1. All other elements are mirrored.

(4) In each step, just as in Table I, the biggest element (or smallest distance) except the main diagonal, is searched. These elements are combined. The distance is memorized and shown in the output as the cluster coefficient for each step.

(5) The elements in the previous step are combined by shrinking the matrix by one dimension. All elements in the corresponding line and column are fused to represent the average angle of both elements. Now the previous step is repeated until the matrix is of size 1.

To correctly read Figure 4, Table II shows all elements of the initial matrix in a numbered order. After combining two elements however, say elements 2 and 6, the new element 2 is already a combination of 2 and 6 ($\overline{26}$) and element 6 is deleted, while all successive elements move forward by one. Therefore, after Step 1, the elements no longer match the numbers of the cluster coefficients. They rather entitle the line and column of the reduction matrix. The similarity values of the clusters after each reduction step can be visualized in a graphical representation.

Figure 3 shows how close the combined elements were after each reduction step that was carried out. This graphic shows a complete cluster analysis, with no termination before the last cluster is built. It shows very tight relations between the first four combinations of the amazon cluster and the first three of the credit card service. Medium distances of 0.4–0.7 can be seen as moderately

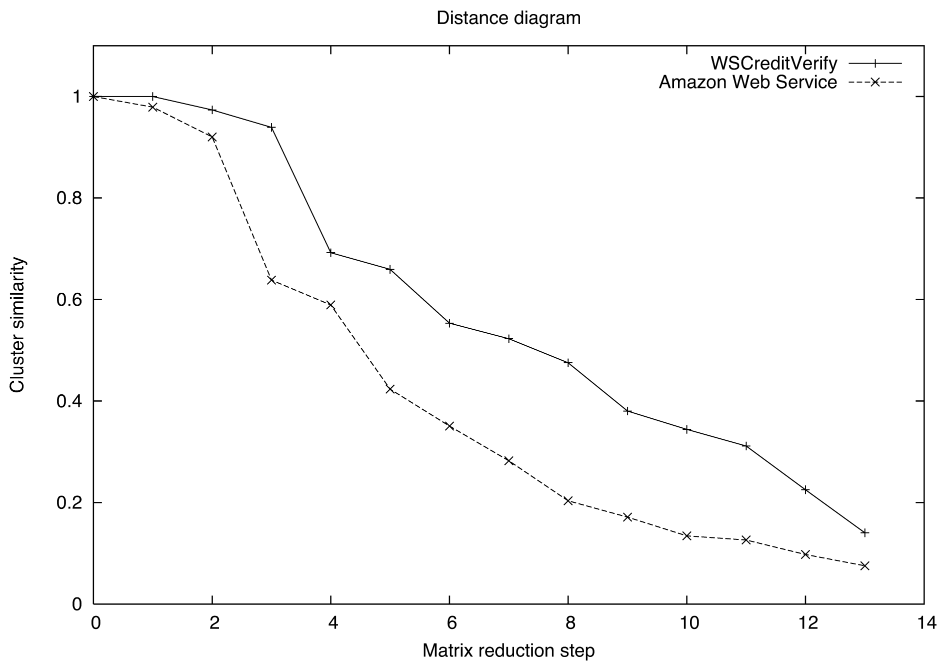

Fig. 3. Cluster similarity diagram.

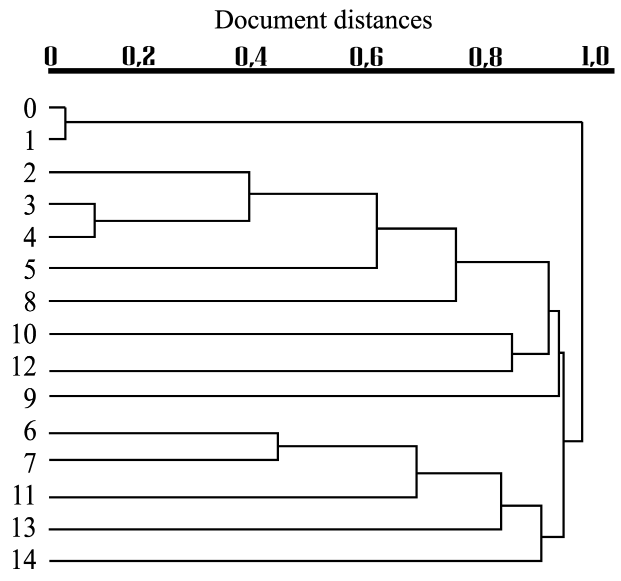

Fig. 4. Dendrogram for "Amazon.wsdl."

relevant, while those below are of low significance. Alternatively, it is possible to define a termination point either by setting a maximum distance a cluster may reach or a maximum number of clusters that should be built. When using a maximum distance, all elements in a more or less tight cluster are considered to be strongly related to each other, while the individual distances are not viewed with special concern. This makes it quite easy visualize cluster results, because individual distance values don't have to be displayed. On the other hand, the result looses a lot of its value when omitting these elements. The decision as to which method to use, or if a complete analysis is preferable to one with an early termination, depends purely on how the result should be presented to the user.

In this case, we decided to use the complete result and visualize it by using a dendrogram. See Figure 4 for an example of the Amazon-Cluster with 15 elements.

When examining the graphic, these four tightly clustered elements can be identified. First is a very tight cluster of Amazon and "ECOWS WS sample," which essentially is a copy of the Amazon WSDL file. This file was injected to ensure that the cluster algorithm puts them in the same cluster with the minimum distance. The next cluster (3 and 4) consists of variations of the Google Web service. Services 2 to 5 are various versions of the same description either gathered from different sources or being different versions, changed by the provider. This cluster is later joined by element 2 at a distance of about 0.4. The first moderately tight cluster is built by "xExchanges" and "Exchanges," both services to query worldwide exchange rates, joined by "InsiderTransactionInfo," also a service for the financial sector.

To summarize, the traditional search methods and information retrieval techniques combined with the proposed clustering approach, can enhance the discovery process of Web services. In this particular test case, a search engine like Google produces a close cluster because both Amazon and Google comprise similar functionality like search and query execution. Other elements, such as the one from the financial sector, are not strongly related to Amazon directly, but their close relation to each other is recognized by the algorithm. From a usability point of view, there are still two possibilities left that have to be considered.

There are essentially two ways to create the initial matrix.

(1) The method used in the example is to execute a query, then select one of the results and do a subsequent query with this vector as the search element. The best n matches are then used to produce the initial matrix.
(2) Another possibility would be to use the result of the initial query to populate the matrix elements without an additional query process. The difference would be that all elements contained in the matrix are directly related to the query string. On the other hand, it would also mean that resulting clusters are not complete, but may actually comprise additional elements.

Both methods are theoretically possible and sound. It is simply a matter of user preference which to actually use. Formally, the former method is the only correct one. The latter can be seen as a tweak that can be applied in a productive environment if required.

5.2 Performance

As mentioned in the earlier sections, it is quite difficult to measure performance without a decent set of elements in the repository. Therefore, the performance

Table III. Performance Comparison

	Repository Size [Files]		
Performance Element	274	549	1096
Cross-query average [ms/element]	167.2	278.1	580.3
Sort time [ms/element]	62	140	282
Keyword retrieval time [ms]	16	15	18
Original query time [ms]	31	53	92
Matrix reduction time [ms]	110	110	110
Overall time for size 15 Matrix [ms]	3750	6112	12322

evaluation is based on an extrapolation of existing elements to demonstrate scalability issues. Furthermore, it has to be kept in mind, that the presented implementation is still a research prototype where the time to optimize performance is limited. The matrix generation process for example, was accelerated to 40% of the original time by introducing proper database indexes and adjusting the queries accordingly. There are still further possibilities, but those enhancements would not be essential for the approach itself. The actual numbers are shown in Table III.

The measures were taken with the same query used in all of the previous examples, with the query string being "amazon web service." For the cluster creation, the matrix size is set to 15 elements.

The cross-query time is used to execute a query that fills one line of the initial matrix and is an average of all 15 queries. It can be seen, that it grows linearly each time the repository size doubles. Directly related to it, is the time used to execute the whole cluster algorithm. This proves that the theoretical assumption that one could execute a cluster algorithm with less than $\mathcal{O}(n^2)$ effort is correct. However, a linear growth is not the best result for the cluster algorithm. When looking at the original query time in the table, the impact of the dimensional reduction becomes obvious. Instead of doubling the time to process a query, the necessary time barely triples for a repository of four times its original size. And there is still room for further enhancements in this direction, by optimizing the generated database queries for instance. At the moment, the query generator is optimized for normal search queries, because the search engine is still the main focus. Because of the heightened possibility of term occurrences in a cluster query, new ways to reduce the query time for whole vectors have to be found. The times for retrieving the keywords of a single vector remain more or less constant because they can be handled in a simple query. Small fluctuations in the exact values are caused by the java timestamp functionality which is limited in its precision and varying speed in the database connection.

Another performance-relevant issue is the persistence mode to which the search engine is set. The volatile form performs much better. Here the queries are not executed on a database but directly on the memory-based hash tables. Nevertheless, this does not affect scalability issues. The measurements and estimations of effort presented in this chapter basically still apply. The change merely effects the concrete execution times. That means, when using the volatile form of the search engine, queries are executed much faster but the grow rate for the computation expense is the same as for the database.

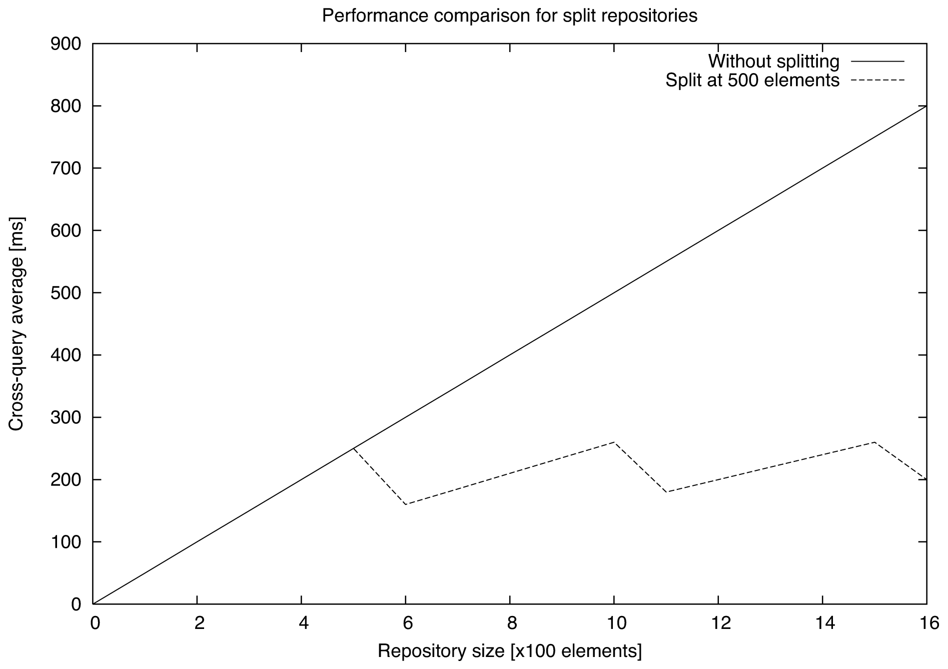

Fig. 5. Performance gain with split repositories.

An important aspect of this research is implied by the general structure of the search engine itself. As already mentioned in the previous chapters of this article, the search engine is designed to work on distributed repositories. That means queries can be processed on the local repository with a single request, or on a compound repository that acts like a large one, consisting of several parts. Because the steps needed to fill the cluster matrix are nothing more than large queries, it is possible to process them the same way as an ordinary distributed query. To do so, the cluster processing has to be moved to the front-end, because this is where the final result list is generated. Our implementation uses the server-side method because it is easier to debug and implement, but as soon as the query processor is optimized for the clustering requests, we will also provide a client-side version for the cluster engine, which is able to take advantage of the original distribution capabilities. As a result, it would be possible to define a maximum time a search or cluster request can take before it has to be split in two separate repositories [Platzer and Dustdar 2005]. The split parts could then be processed in parallel, which increases scalability and of course performance. When looking at the values from our server-side evaluation in Table III it becomes obvious that the cross-query time acts as a bottleneck for the whole approach. In Figure 5 we visualized the performance gain for the cross-query times when the distribution capabilities are enabled. The partitioning was such that repositories are split once they reach 500 elements. Then an empty vector space is created and all elements are evenly distributed. The resulting federation is connected via a Web service. The queries for the matrix generation were run on the federation, which produces the result in a fraction of the original time because of the parallel processing capabilities. A small overhead of 11 ms in the processing time occurs because of the additional effort to merge the results once they return from each peer. This method of speeding up the whole process allows one to define an upper bound for the execution time of both the original queries and the cluster algorithm as a whole.

In a nutshell, the performance of our implemented approach lives up to the expectations and the original concept. At the same time, there is still the possibility for performance enhancements and tweaks, especially where database access and query generation are concerned.

6. RELATED WORK

Search and search-related aspects of Web services are highly investigated fields throughout the service oriented community. In most cases, Web service discovery is the driving force behind the research. The reason is simply that this particular area still raises some very interesting issues that need to be addressed. Service discovery in ad hoc networks that are based on Web services [Friedman 2002] for instance, has to deal with some very special problems regarding information propagation and centralization. It is basically the same problem that arises with all P2P networks. The highly fluctuating nature of such designs would cause there to be a single point of failure so they would have to encompass a feature to suitably propagate service information. Ordinary methods for service discovery, like UDDI registries most certainly fail in such environments because they are not adaptable enough. Furthermore, they lack the very important feature of joining service repositories, which is necessary for federations of Web services. This particular issue was addressed by Sivashanmugam et al. [2004], for example. Here, the authors specifically dealt with this problem and developed a discovery mechanism that allows a user to find services in a federated service environment. The system is called MWSDI and relies on a decentralized structure without a central component. The general layout of our discovery approach allows a setup that is independent of the underlying structure. It can work in centralized environments as part of an ordinary services registry. At the same time it is possible to deploy the search engine in a federated environment as part of an ad hoc network.

Other work focuses not on the principle structure of the service infrastructure, but on the quality of the search result itself. Almost every registry ever built encompasses some sort of search facility to pick up contained services. In most cases the search functionality is implemented by a full text search on the underlying database. In other words, searching those repositories is not always concerned as particularly important, let alone a convenient and more powerful way to relate search results and repository content. Those topics seem to have gained attention only in recent years. In Caverlee et al. [2004], a search engine named BASIL is introduced that tries to relate Web services by using bias-based techniques. Here, the repositories are supposed to encompass only data-intensive services like searching for DNA sequences. The similarity measure is based on the exchanged documents and is therefore a personalized type of search approach. Some of the techniques used are similar to the work presented in Platzer and Dustdar [2005] with the difference that the weighting is not performed at runtime, but in advance. Furthermore, the approaches work on different repository structures.

Dong et al. [2004] describe a clustering-based search approach for Web services, which is implemented in their search engine, Woogle. Their search approach consists of two main phases: first, the user issues a query at the search engine by specifying a set of keywords. The search engine returns a list of Web services that match that query. Second, their tool extracts a set of semantic concepts using natural language descriptions that have to be provided by the Web service in the repository. To do so, the authors exploit the co-occurrence of terms in Web service inputs and outputs, of operations to cluster terms into meaningful concepts by applying an agglomerative clustering algorithm. This makes it possible to combine the original keywords with the extracted concepts and compare two services on a keyword and concepts level to improve precision and recall. This approach leads to significantly better results than a plain keyword-based search. Compared to our approach, this technique requires a previously built index, before it can take advantage of the additional information. Although it is possible to work with real-world services as they exist today, it requires a certain level of preprocessing, thus limiting the scalability.

In Abramowicz et al. [2007], the authors propose an architecture for Web services filtering and clustering. The service filtering mechanism is based on user and application profiles that are described using OWL-S (Web Ontology Language for Services). The effectiveness of the filters is based on a clustering analysis that compares services related clusters. The objectives of this matchmaking process are to save execution time and to improve the refinement of the stored data. Similarly to the previously used method, this technique relies on the existence of semantically attached Web service descriptions (SAWSDL) to enable the methodology. We believe this to be a limited approach, at least when it is seen in today's environment. SAWSDL is practically not used in available service descriptions. Furthermore, when requiring semantic information to be available a priori, the technique of clustering is no longer necessary. A domain-specific ontology already contains an intrinsic classification system, eliminating the necessity of achieving the same effect with clustering techniques.

Hess et al. [2004] propose an approach and a tool called ASSAM, which allows developers to create semantic information (metadata) for existing services. Their approach uses an iterative machine learning algorithm that treats Web service information such as operations, inputs, and outputs, as a text classification problem. The tool learns from training data, that is, existing Web services with semantic information. The authors classify services along different attributes: category (e.g., weather information), domain (e.g., search weather by ZIP code) and datatype (ZIP code). Additionally, documents and document parts are used to refer to a WSDL as a whole or to parts of it (e.g., operations, inputs or outputs). The evaluation shows that a machine learning algorithm assists developers to create semantic annotations if certain assumptions can be made. It is the approach most closely related to ours, however the most obvious difference is that no training set is required for our method. Such a training set essentially represents an ontology, even if it may differ in its implementation. Although we believe that the results produced will be of greater value once the training is sufficient, our method does not require any human interaction whatsoever to complete the task.

Some other aspects of service discovery are discussed in Ran [2003]. Here the authors propose an extension of the current UDDI infrastructure to add QoS descriptions for a given service. Due to the relative openness of the UDDI technology, this approach can actually work with existing technologies. On the other hand, this openness is sometimes seen as one of the major reasons that UDDI has failed to dominate the public Web service domain. In Benatallah et al. [2005], a rather formal approach is presented, where semantic annotations are utilized for automated service discovery. Apart from the discovery issues, the description logic could also be used to formally describe interservice relationships. Those relationships can reach from input/output matching on a syntactical level, up to QoS descriptions of whole compositions. Yu et al. [2007] propose a method to select services based on their quality. The selection algorithm assumes services where QoS parameters are already described and focuses on optimizing an end-to-end composition with respect to the overall QoS.

Unlike general clustering problems, as discussed by Andritsos and Tzerpos [2003] for example, Web service indexes usually deal with a structured data cloud, which enables an appliance of different and more specialized clustering methods. This difference also applies to Web page clustering as used by today's Web page search engines. In the previously mentioned work, the goal is to visualize the structure of a software component and therefore even help reverse engineer it. The algorithms used are highly sophisticated and are targeted to discover structural dependencies. In our article however, the index structure is known a priori.

There is plenty of reference material available for the statistical background. The theoretical foundation for creating this kind of relationship is best described with statistical cluster analysis [Eckey et al. 2002]. These methods are used in various research fields to describe similarities of metric values of all kinds. The main problem faced here is the high complexity of ordinary methods. The usually exponential growth rate enormously limits its capabilities for high dimensional data structures. To cope with this problem, la Torre and Kanade [2006] introduced a modified cluster algorithm that performs dimensional reduction and clustering at the same time. The method is designed to increase performance in large vector spaces with 1000 and more dimensions. Even though the application area is different in this work, some of the ideas are built on a common ground. As compared with our work, a whole data cloud with a strong cluster layout has to be processed to categorize the single elements, while we deal with the issue of finding possible clusters for a particular query. Furthermore, we apply dimensional reduction before the original cluster algorithm to speed up the matrix generation rather than increase the performance of the algorithm itself.

7. CONCLUSIONS

In this article, we presented an approach to create clusters of Web services to allow Web service consumers to easily find and relate specific services to a given query. The theoretical basis is formed by statistical methods of cluster analysis in n-dimensional vector spaces with numerical characteristics. The implementation shows that the approach is feasible and even exceeds our expectations in certain respects. The implementation of the cluster engine shows that the possibilities for the processing power do not end at $\mathcal{O}(n)$ expense. It is quite the opposite. By using the developed method, the query processing speed is the only limitation for the whole method. By further enhancing the query process for complete vectors, it is theoretically possible to enhance the cluster speed even beyond $\mathcal{O}(n)$ in the future. In the best case, the growth rate converges to the same amount as is currently possible for the original search. This assumption is based on the observation that common queries perform better due to the dimensional reduction that can be applied when retrieving related documents. A similar method to reduce the query time for the matrix generation has to be implemented to reduce the processing time. Furthermore, an optimization of the generated database queries for the specific form of request issued by the cluster algorithm can further enhance performance. Without altering the query algorithm though, the gain will be proportional and is, therefore, not considered with the utmost priority for the future.

One of the conceptual issues worth discussing is the entry point for the cluster algorithm. In the prototype a query is first issued with the vector of one result element to fill the cluster matrix. This starting point is selected by the Web interface.

Alternatively, it is also possible to fill the matrix with the results of the original query itself. As an effect, the cluster matrix would not represent the n most related elements compared to one WSDL file but the nearest matches for the query and their relations to each other. It can be seen as a thinner populated space where the original result vectors are highlighted. The remaining vectors are still necessary to build the right vector relations, otherwise a reduced vector space with n elements for an n-sized matrix could be used. This would certainly speed up the processing time, but it is not possible without loosing important term information. Although formally not complete, this cluster might provide a better understanding of the document relationships, and therefore be of greater value to the user than a cluster analysis based on a complete vector. This issue is at least worth investigating and will be part of our future tasks.

Finally, the presented indexing method opens the opportunity to forswitching from keyword-based indexes to quality-based ones. The current method ensures the provision of an unbiased representation of the contained services. The aim of this article is to provide natural language-enabled query processing that will be the preferred method for most cases. In some cases though, it could be necessary to search for certain quality elements of a service. The approach presented in this works for both query methods. To extend the capabilities to quality information, the various Quality of Service (QoS) parameters recognized by the search engine have to be properly defined and evaluated. In

Rosenberg et al. [2006], we introduced a framework to evaluate such parameters and implemented a prototype capable of automatically invoking previously unknown services to measure nonfunctional attributes like availability, latency and so forth, on the TCP level. These values can be used for indexing services with quality-based vectors. Furthermore, other metadata like location information, can be used to further enrich the service vector and therefore leverage the discovery capability from a purely syntactic to a quality-enabled level.

To summarize, we believe that the work presented in this article is one important, currently missing, element in the course of design for a usable system for service discovery and Web service relationship management.

ACKNOWLEDGMENTS

Our sincere thanks go to Johann Oberleitner for his help with the optimization of the generated queries that are necessary to process the cluster elements. Thanks to the reviewers for their appreciated suggestions and help to improve the article.

REFERENCES

ABRAMOWICZ, W., HANIEWICZ, K., KACZMAREK, M., AND ZYSKOWSKI, D. 2007. Architecture for Web services filtering and clustering. In *Proceedings of the International Conference on Internet and Web Applications and Services (ICIW)*. 18.

ANDRITSOS, P. AND TZERPOS, V. 2003. Software clustering based on information loss minimization. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*. IEEE Computer Society, Washington, DC.

BENATALLAH, B., HACID, M.-S., LEGER, A., REY, C., AND TOUMANI, F. 2005. On automating Web services discovery. *Int. J. VLDB 14*, 1 (3), 84–96.

CAVERLEE, J., LIU, L., AND ROCCO, D. 2004. Discovering and ranking Web services with BASIL: A personalized approach with biased focus. In *Proceedings of the 2nd International Conference on Service-Oriented Computing (ICSOC'04)*. ACM Press, 153–162.

CHRISTENSEN, E., CURBERA, F., MEREDITH, G., AND WEERAWARANA, S. 2001. Web Services Description Language (WSDL) 1.1. W3C. http://www.w3.org/TR/wsdl.

DONG, X., HALEVY, A. Y., MADHAVAN, J., NEMES, E., AND ZHANG, J. 2004. Simlarity search for Web services. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB'04)*. 372–383.

ECKEY, H.-F., KOSFELD, R., AND RENGERS, M. 2002. *Multivariate Statistics*. Gabler.

FRIEDMAN, R. 2002. Caching Web services in mobile ad hoc networks: Opportunities and challenges. In *Proceedings of the 2nd ACM International Workshop on Principles of Mobile Computing (POMC'02)*. ACM Press, 90–96.

HESS, A., JOHNSTON, E., AND KUSHMERICK, N. 2004. ASSAM: A tool for semi-automatically annotating semantic Web services. In *Proceedings of the 3rd International Semantic Web Conference (ISWC'04)*. 320–334.

IBM. 2005. IBM business registry. https://uddi.ibm.com/ubr/registry.html.

LA TORRE, F. D. AND KANADE, T. 2006. Discriminative cluster analysis. In *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*. ACM Press, 241–248.

MACQUEEN, J. B. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*. vol. 1. University of California Press, Berkeley, 281–297.

MAGDALENIC, I., VRDOLJAKAND, B., AND SKOCIR, Z. 2006. Towards dynamic Web service generation on demand. In *Proceedings of the International Conference on Software in Telecommunications and Computer Networks, (SoftCOM'06)*.

MICROSOFT. 2005. Microsoft public uddi registry. http://uddi.microsoft.com/inquire.

OASIS 2005. Universal Description, Discovery and Integration (vol.) 3.0 (UDDI) Specification. OASIS. http://www.oasis-open.org/committees/uddi-spec.

Papazoglou, M. P. 2003. Service-oriented computing: Concepts, characteristics and directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering*. 3–12.

Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. 2006. Service-Oriented Computing Research Roadmap. http://infolab.uvt.nl/pub/papazogloump-2006-96.pdf

Platzer, C. 2007. V.U.S.E. - The Vector Space Web Service Search Engine. http://vuse.de.vu/.

Platzer, C. and Dustdar, S. 2005. A vector space search engine for Web services. In *Proceedings of the 3rd European IEEE Conference on Web Services (ECOWS'05)*.

Ran, S. 2003. A model for Web services discovery with QoS. *SIGecom Exch. 4*, 1, 1–10.

Rosenberg, F., Platzer, C., and Dustdar, S. 2006. Boot-strapping performance and dependability attributes of Web services. In *Proceedings of the IEEE Conference on Web Services (ICWS'06)*, 205–212.

Sivashanmugam, K., Verma, K., and Sheth, A. 2004. Discovery of Web services in a federated registry environment. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, 270–278.

W3C. 2000. Resource Description Framework (RDF). http://www.w3.org/RDF.

Yu, T., Zhang, Y., and Lin, K.-J. 2007. Efficient algorithms for Web services selection with end-to-end qos constraints. *ACM Trans. Web 1*, 1, 6.

Received October 2008; accepted March 2009