Kvasir: Seamless Integration of Latent Semantic Analysis-Based Content Provision into Web Browsing

Liang Wang Sotiris Tasoulis Teemu Roos Jussi Kangasharju
Department of Computer Science
University of Helsinki, Finland
{firstname.lastname}@cs.helsinki.fi

ABSTRACT

The Internet is overloading its users with excessive information flows, so that effective content-based filtering becomes crucial in improving user experience and work efficiency. Latent semantic analysis has long been demonstrated as a promising information retrieval technique to search for relevant articles from large text corpora. We build Kvasir, a semantic recommendation system, on top of latent semantic analysis and other state-of-art technologies to seamlessly integrate an automated and proactive content provision service into web browsing. We utilize the processing power of Apache Spark to scale up Kvasir into a practical Internet service. Herein we present the architectural design of Kvasir, along with our solutions to the technical challenges in the actual system implementation.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*clustering, information filtering*

1. INTRODUCTION

Currently, the Internet is overloading its users with excessive information flows. Therefore, smart content provision and recommendation become more and more crucial in improving user experience and efficiency in using Internet applications. For a typical example, many users are most likely to read several articles on the same topic while surfing on the Web. Hence many news websites (e.g., The New York Times, BBC News and Yahoo News) usually group similar articles together and provide them on the same page so that the users can avoid launching another search for the topic. However, most of such services are constrained within a single domain, and cross-domain content provision is usually achieved by manually linking to the relevant articles on different sites. Meanwhile, companies like Google and Microsoft take advantage of their search engines and provide customizable keywords filters to aggregate related articles across various domains for users to subscribe. However, to subscribe a topic, a user needs to manually extract keywords from an article, then to switch between different search services while browsing the web pages.

In general, seamless integration of intelligent content provision into web browsing at user interface level remains an open research question. No universally accepted optimal design exists. Herein we propose Kvasir[1], a system built on top of latent semantic analysis (LSA). We show how Kvasir can be integrated with the state-of-art technologies (e.g., Apache Spark, machine learning, etc.). Kvasir automatically looks for the similar articles when a user is browsing a web page and injects the search results in an easily accessible panel within the browser view for seamless integration. The ranking of the results is based on the cosine similarity in LSA space, which was proposed as an effective information retrieval technique almost two decades ago [6]. Despite some successful applications in early information systems, two technical challenges practically prevent LSA from becoming a scalable Internet service. First, LSA relies on large matrix multiplications and singular value decomposition (SVD), which become notoriously time and memory consuming when the document corpus is huge. Second, LSA is a vector space model, and fast search in high dimensional spaces tends to become a bottle-neck in practice.

We must emphasize that Kvasir is not meant to replace the conventional web search, recommender systems, or other existing technologies discussed in Section 2. Instead, Kvasir represents a potential solution to enhancing user experience in future Internet applications. In this paper, by presenting the architectural components, we show how we tackle the scalability challenges confronting Kvasir in building and indexing high dimensional language database. To address the challenge in constructing the database, we adopt a rank-revealing algorithm for dimension reduction before actual SVD. To address the challenge in high dimensional search, we utilize approximate nearest neighbor search to trade off accuracy for efficiency. The corresponding indexing algorithm is optimized for parallel implementation.

Specifically, our contributions are: (1) we present the architecture of Kvasir, which is able to seamlessly integrate LSA-based content provision in web browsing by using state-of-art technologies. (2) we implement the first stochastic SVD on Apache Spark, which can efficiently build LSA-based language models on large text corpora. (3) we propose a parallel version of the randomized partition tree algorithm which provides fast indexing in high dimensional vector spaces using Apache Spark.[2]

[1]Kvasir is the acronym for *Knowledge ViA Semantic Information Retrieval*, it is also the name of a Scandinavian god in Norse mythology who travels around the world to teach and spread knowledge and is considered extremely wise.

[2]The source code of the key components in Kvasir are publicly accessible and hosted on Github. We will release all the Kvasir code after the paper is accepted.

2. BACKGROUND AND RELATED WORK

There are several parallel efforts in integrating intelligent content provision and recommendation in web browsing. They differentiate between each other by the main technique used to achieve the goal. The initial effort relies on the semantic web stack proposed in [5], which requires adding explicit ontology information to all web pages so that ontology-based applications (e.g., Piggy bank [20]) can utilize ontology reasoning to interconnect content semantically. Though semantic web has a well-defined architecture, it suffers from the fact that most web pages are unstructured or semi-structured HTML files, and content providers lack of motivation to adopt this technology to their websites. Collaborative Filtering (CF) [8, 24], which was first coined in Tapestry [16], is a thriving research area and also the second alternative solution. Recommenders built on top of CF exploit the similarities in users' rankings to predict one user's preference on a specific content. CF attracts more research interest these years due to the popularity of online shopping (e.g., Amazon, eBay, Taobao, etc.) and video services (e.g., YouTube, Vimeo, Dailymotion, etc.). However, recommender systems need user behavior rather than content itself as explicit input to bootstrap the service, and is usually constrained within a single domain. Cross-domain recommenders [11, 25] have made progress lately, but the complexity and scalability need further investigation. Search engines can be considered as the third alternative though a user needs explicitly extract the keywords from the page then launch another search. The ranking of the search results is based on link analysis on the underlying graph structure of interconnected pages (e.g., PageRank [30] and HITS [23]). As the fourth alternative, Kvasir takes another route by utilizing information retrieval (IR) [14, 27]. Kvasir belongs to the content-based filtering and emphasizes the semantics contained in the unstructured web text. In general, a text corpus is transformed to the suitable representation depending on the specific mathematical models (e.g., set-theoretic, algebraic, or probabilistic models), based on which a numeric score is calculated for ranking. Context awareness is the most significant advantage in IR, which has been integrated into Hummingbird – Google's new search algorithm.

Inside Kvasir, the design covers a wide range of different topics, each topic has numerous related work. In the following, we constrain the discussion only on the core techniques used in the system. Due to the space limit, we cannot list all related work, we recommend using the references mentioned in this section as a starting point for further reading. The initial idea of using linear algebraic technique to derive latent topic model was proposed in [6]. As the core operation of LSA, SVD is a well-established subject and has been intensively studied over three decades. Recent efforts have been focusing on efficient incremental updates to accommodate dynamic data streams [7] and scalable algorithms to process huge matrices.

Efficient nearest neighbor search in high dimensional spaces has attracted a lot of attention in machine learning community. There is a huge body of literature on this subject which can be categorized as graph-based [17, 35], hashing-based [19, 34, 36], and partition tree-based solutions [12, 22, 28,32]. The graph-based algorithms construct a graph structure by connecting a point to its nearest neighbors in a data set. These algorithms suffer from the time-consuming construction phase. As the best known hashing-based solution, locality-sensitive hashing (LSH) [4,26] uses a large number of hash functions with the property that the hashes of nearby points are also close to each other with high probability. The performance of a hashing-based algorithm highly depends on the quality of the hashing functions, and it is usually outperformed by partition tree-based methods in practice [28]. In particular the Randomized Partition tree (RP-tree) [12] method have been shown to be very successful in practice, while it was also recently shown [13] that its probability to fail is bounded when the data are documents from a topic model. RP-tree was initially introduced as an improvement over the k-d tree method that is more appropriate for use in high dimensional spaces, drawing inspiration from LSH [26]. In this work, inspired by a recent application of the random projection method [33], we take advantage of the simplicity of the RP-tree method to further develop its parallel version.

There are also abundant software toolkits with different emphasises on machine learning and natural language processing. We only list the most relevant ones: scikit-learn [3], FLANN [29], Gensim [31], and ScalaNLP [2]. scikit-learn includes many general-purpose algorithms for data mining and analysis, but the toolkit is only suitable for small and medium problems. Gensim and ScalaNLP have a clear focus on language models. ScalaNLP's linear algebra library (Breeze) is not yet mature enough, which limits its scalability. On the other hand, Gensim scales well on large corpora using a single machine, but fails to provide efficient indexing and searching. Though FLANN provides fast nearest neighbor search, it requires loading the full data set in to memory therefore severely limits the problem size [21]. None of the aforementioned toolkits provides a horizontally scalable solution on big data frameworks. The default machine learning library MLlib in Apache Spark misses the stochastic SVD and effective indexing algorithms [37].

3. KVASIR ARCHITECTURE

At the core, Kvasir implements an LSA-based index and search service, and its architecture can be divided into two subsystems as *frontend* and *backend*. Figure 1 illustrates the general workflow and internal design of the system. The frontend is currently implemented as a lightweight extension in Chrome browser. The browser extension only sends the page URL back to the KServer whenever a new tab/window is created. The KServer running at the backend retrieves the content of the given URL then responds with the most relevant documents in a database. The results are formatted into JSON strings. The extension presents the results in a friendly way on the page being browsed. From user perspective, a user only interacts with the frontend by checking the list of recommended articles that may interest him.

To connect the frontend, the backend exposes one simple *REST API* as below, which gives great flexibility to all possible frontend implementations. By loosely coupling with the backend, it becomes easy to mash-up new services on top of Kvasir. Line 1 and 2 give an example request to Kvasir service. `type=0` indicates that `info` contains a URL, otherwise `info` contains a piece of text if `type=1`. Line 4-9 present an example response from the server, which contains the metainfo of a list of similar articles. Note that

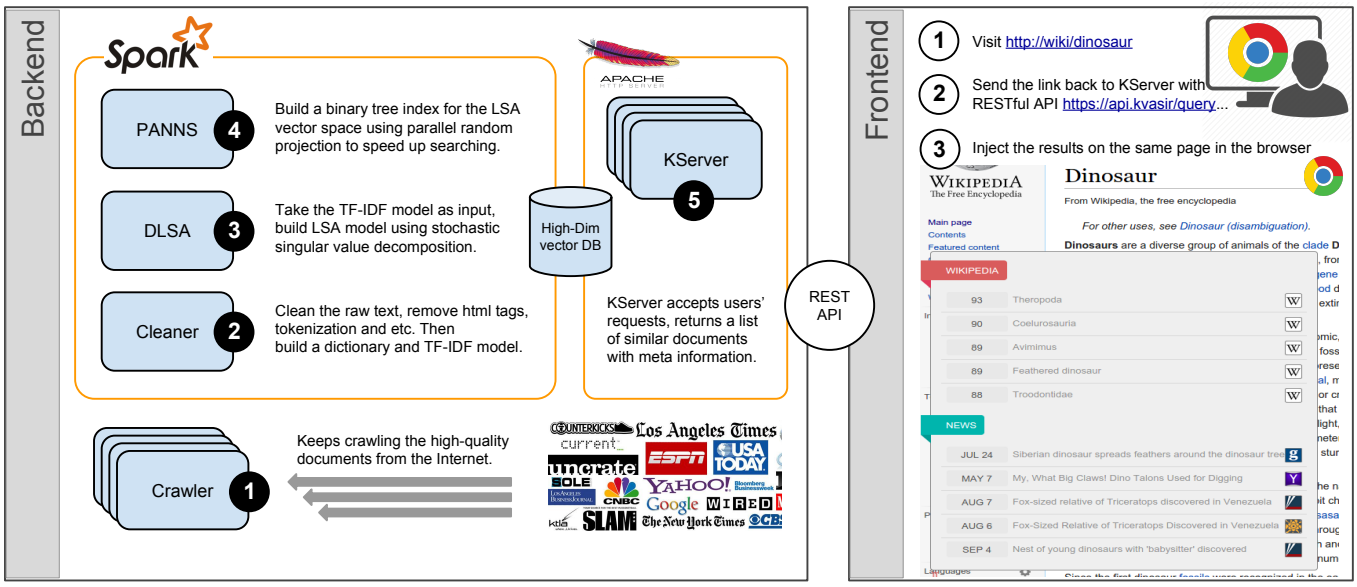

Figure 1: Kvasir architecture – there are five major components in the backend system, and they are numbered based on their order in the workflow. Frontend is implemented in a Chrome browser, and connects the backend with a RESTful API.

the frontend can refine or rearrange the results based on the metainfo (e.g., similarity or timestamp).

```
POST
https://api.kvasir/query?type=0&info=url

{"results": [
  {"title": document title,
   "similarity": similarity metric,
   "page_url": link to the document,
   "timestamp": document create date}
]}
```

The backend system implements indexing and searching functionality which consist of five components: Crawler, Cleaner, DLSA, PANNS and KServer. Three components (i.e., Cleaner, DLSA and PANNS) are wrapped into one library since all are implemented on top of Spark. The library covers three phases as text cleaning, database building, and indexing. We briefly present the main tasks in each component as below.

Crawler collects raw documents from the Web then compiles into two data sets. One is the English Wikipedia dump, and another is compiled from over 300 news feeds of the high-quality content providers. Table 1 summarizes the basic statistics of the data sets. Multiple instances of the Crawler run in parallel on different machines. Simple fault-tolerant mechanisms like periodical backup have been implemented to improve the robustness of crawling process. In addition to the text body, the Crawler also records the timestamp, URL and title of the retrieved news as metainfo, which can be further utilized to refine the search results.

Cleaner cleans the unstructured text corpus and converts the corpus into term frequency-inverse document frequency (TF-IDF) model. In the preprocessing phase, we clean the text by removing HTML tags and stopwords, deaccenting, tokenization, etc. The dictionary refers to the vocabulary of a language model, its quality directly impacts the model performance. To build the dictionary, we exclude both extremely rare and extremely common terms, and keep 10^5 most popular ones as *features*. More precisely, a term is considered as rare if it appears in less than 20 documents, while a term is considered as common if it appears in more than 40% of documents.

DLSA builds up an LSA-based model from the previously constructed TF-IDF model. Technically, the TF-IDF itself is already a vector space language model. The reason we seldom use TF-IDF directly is because the model contains too much noise and the dimensionality is too high to process efficiently even on a modern computer. To convert a TF-IDF to an LSA model, DLSA's algebraic operations involve large matrix multiplications and time-consuming SVD. Since MLlib is unable to perform SVD on a data set of 10^5 features with limited RAM, we implemented our own stochastic SVD on Spark using rank-revealing technique. Section 4.1 discusses DLSA in details.

PANNS builds the search index to enable fast k-NN search in high dimensional LSA vector space. Though dimensionality has been significantly reduced from TF-IDF (10^5 features) to LSA (10^3 features), k-NN search in a 10^3-dimension space is still a great challenge. Naive linear search using one CPU takes over 6 seconds to finish in a database of 4 million entries, which is unacceptably long for any realistic services. PANNS implements a parallel RP-tree algorithm which makes a reasonable tradeoff between accuracy and efficiency. Section 4.2 presents PANNS in details.

KServer runs within a web server, processes the users requests and replies with a list of similar documents. KServer uses the index built by PANNS to perform fast search in the database. The ranking of the search results is based on the cosine similarity metric. A key performance metric for KServer is the service time. We deployed multiple KServer instances on different machines and implemented a simple round-robin mechanism to balance the request loads.

Data set	# of entries	Raw text size	Article length
Wikipedia	3.9×10^6	47.0 GB	Avg. 782 words
News	3.3×10^5	1.4 GB	Avg. 648 words

Table 1: Two data sets are used in the evaluation. Wikipedia represents relatively static knowledge, while News represents constantly changing dynamic knowledge.

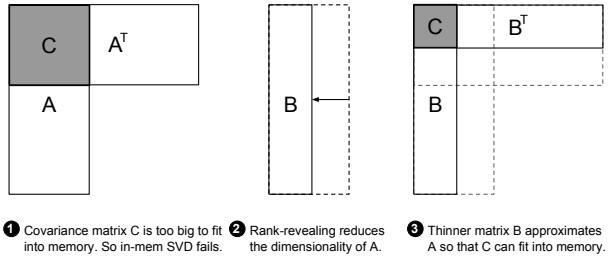

Figure 2: DLSA uses rank-revealing to effectively reduce dimensionality to perform in-memory SVD.

4. PROPOSED ALGORITHMS

Due to the space limit, we only sketch out the key ideas in DLSA and PANNS algorithms rather than present the code line by line. Since the code of the key components in Kvasir are publicly accessible on Github, those who have interest can read the code to further study the algorithmic details [1].

4.1 Distributed Stochastic SVD

The vector space model belongs to algebraic language models, where each document is represented with a row vector. Each element in the vector represents the weight of a term in the dictionary calculated in a specific way. E.g., it can be simply calculated as the frequency of a term in a document, or slightly more complicated TF-IDF. The length of the vector is determined by the size of the dictionary (i.e., number of features). A text corpus containing m documents and a dictionary of n terms will be converted to an $A = m \times n$ row-based matrix. Informally, we say that A grows taller if the number of documents (i.e., m) increases, and grows fatter if we add more terms (i.e., n) in the dictionary. LSA utilizes SVD to reduce n by only keeping a small number of linear combinations of the original features. To perform SVD, we need calculate the covariance matrix $C = A^T \times A$, which is a $n \times n$ matrix and is usually much smaller than A.

We can easily parallelize the calculation of C by dividing A into k smaller chunks of size $[\frac{m}{k}] \times n$, so that the final result can be obtained by aggregating the partial results as $C = A^T \times A = \sum_{i=1}^{k} A_i^T \times A_i$. However, a more serious problem is posed by a large number of columns, n. The SVD function in MLlib is only able to handle tall and thin matrices up to some hundreds of features. For most of the language models, there are often hundreds of thousands features (e.g., 10^5 in our case). The covariance matrix C becomes too big to fit into the memory, hence the native SVD operation in MLlib of Spark fails as the first subfigure of Figure 2 shows.

In linear algebra, a matrix can be approximated by another matrix of lower rank while still retaining approximately properties of the matrix that are important for the problem at hand. In other words, we can use another thinner matrix B to approximate the original fat A. The corresponding technique is referred as rank-revealing QR estimation [18]. A TF-IDF model having 10^5 features often contains a lot of redundant information. Therefore, we can effectively thin the matrix A then fit C into the memory. Figure 2 illustrates the algorithmic logic in DSLA, which is essentially a distributed stochastic SVD implementation.

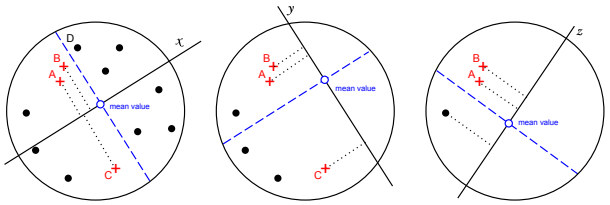

Figure 3: We can continuously project the points on random vectors and use mean value to divide the space for clustering.

4.2 Parallel Randomized Partition Tree

With an LSA model at hand, finding the most relevant document is equivalent to finding the nearest neighbors for a given point in the derived vector space. The distance is usually measured with the cosine similarity of two vectors. However, neither naive linear search nor conventional k-d tree is capable of performing efficient search in such high dimensional space even though the dimensionality has been significantly reduced from 10^5 to 10^3 by LSA.

Nonetheless, we need not locate exact nearest neighbors in practice. In most cases, slight numerical error (reflected in the language context) is not noticeable at all, i.e., the returned documents still look relevant from the user's perspective. By sacrificing some accuracy, we can obtain a significant gain in search speed. The general idea of RP-tree algorithm used here is clustering the points by partitioning the space into smaller subspaces recursively. Technically, this can be achieved by tree-based algorithms. Given a tree built from a database, we answer a nearest neighbor query q in an efficient way, by moving q down the tree to its appropriate leaf cell, and then return the nearest neighbor in that cell. However in several cases q's nearest neighbor may well lie within a different cell. Figure 3 gives a naive example on a 2-dimension vector space. First, a random vector x is drawn and all the points are projected onto x. Then we divide the whole space into half at the mean value of all projections (i.e., the blue circle on x) to reduce the problem size. For each new subspace, we draw another random vector for projection, and this process continues recursively until the number of points in the space reaches the predefined threshold on cluster size. We can construct a binary tree to facilitate the search. As we can see in the first subfigure of Figure 3, though the projections of A, B, and C seem close to each other on x, C is actually quite distant from A and B. However, it can be shown that such misclassifications become arbitrarily rare as the iterative procedure continues by drawing more random vectors and performing splits. For example, in Figure 3, y successfully separates C from A and B.

Another kind of misclassification is that two nearby points are unluckily divided into different subspaces, e.g., B and D in the left panel of Figure 3. To get around this issue, the

authors in [26] proposed a tree structure (spill tree) where each data point is stored in multiple leaves, by following overlapping splits. Although the query times remain essentially the same, the required space is significantly increased. In this work we choose to improve the accuracy by building multiple RP-trees. However, in this case one would need to store a large number of random vectors, introducing significant storage overhead as well. For a corpus of 4 million documents, if we use 10^5 random vectors (i.e., a cluster size of 20), and each vector is a 10^3-dimension real vector (32-bit float number), the induced storage overhead is about 381.5 MB for each RP-tree.

One possible solution to reduce the index size is reusing the random vectors. Namely, we generate a pool of random vectors once, then randomly choose one from the pool each time one is needed. However, the immediate challenge emerges when we try to parallelize the tree building on multiple nodes. Because we need broadcast the pool of vectors onto every node, which causes significant network traffic.

To address this challenge, we propose to use a pseudo random seed in building and storing search index. Instead of maintaining a pool of random vectors, we just need a random seed for each RP-tree. The computation node can build all the random vectors on the fly from the given seed. From the model building perspective, we can easily broadcast several random seeds instead of a large matrix in the network, therefore we improve the computation efficiency. From the storage perspective, we only need store one 4-byte random seed for each RP-tree. In such a way, we successfully reduce the storage overhead from 47.7 GB to 512 B for a search index consisting of 128 RP-trees (with cluster size 20).

4.3 Caching to Scale Up

Even though our indexing algorithm is able to significantly reduce the index size, the index will eventually become too big to fit into memory when the corpus grows from millions to trillions of documents. One engineering solution is using MMAP provided in operating systems which maps the whole file from hard-disk to memory space without actually loading it into the physical memory. The loading only happens when a specific chunk is accessed. Loading and eviction are handled automatically by the operating system.

Search performance may degrade if the access is truly random on a huge index. In practice, this is highly unlikely since the pattern of user requests follows a clear Zipf-like distribution. In other words, (i) most users are interested in a relatively small amount of articles; (ii) most of the articles that an individual user is reading at any given time are similar. These two observations imply that only a small part of the index trees is frequently accessed at any given time, which leads to the actual performance being much better than that of a uniformly distributed access pattern.

5. PRELIMINARY EVALUATION

Because scalability is the main challenge in Kvasir, the evaluation revolves around two questions: (i) how fast we can build a database from scratch using the library we developed for Apache Spark; (ii) how fast the search function in Kvasir can serve users' requests. In the following, we present the results of our preliminary evaluation.

# of CPUs	Cleaner	DLSA	PANNS	Total
1	1.32	20.23	13.99	35.54
5	0.29	6.14	2.86	9.29
10	0.19	4.22	1.44	5.85
15	0.17	3.14	0.98	4.29
20	0.16	2.61	0.77	3.54

Table 2: The time needed (in hours) for building an LSA-based database from Wikipedia raw text corpus. The time is decomposed to component-wise level. Search index uses 128 RP-trees with cluster size of 20.

The evaluation is performed on a small testbed of 10 Dell PowerEdge M610 nodes. Each node has 2 quad-core CPUs, 32GB memory, and is connected to a 10-Gbit network. All the nodes run Ubuntu SMP with a 3.2.0 Linux kernel. ATLAS (Automatically Tuned Linear Algebra System) is installed on all the nodes to support fast linear algebra operations. Three nodes are used for running Crawlers, five for running our Spark library, and the rest two for running KServer to serve users' requests as web servers. Due to the space limit, we only report the results on using Wikipedia data set. News data set leads to consistently better performance due to its smaller size.

5.1 Database Building Time

We evaluate the efficiency of the backend system using our Spark library, which includes text cleaning, model building, and indexing the three phases. We first perform a sequential execution on a single CPU to obtain a benchmark. With one CPU, it takes over 35 hours to process the Wikipedia data set. Using 5 CPUs to parallelize the computation, it takes about 9 hours which is almost 4 times improvement. From Table 2, we can see that total building speed is improved sublinearly. The reason is because the overhead from I/O and network operations eventually replace CPU overhead and become the main bottleneck in the system.

By zooming in the time usage and checking the component-wise overhead, DSLA contributes most of the computation time while Cleaner contributes the least. Cleaner's tasks are easy to parallelize due to its straightforward structure, but there are only marginal improvements after 10 CPUs since most of the time is spent in I/O operations and job scheduling. For DLSA, the parallelism is achieved by dividing a tall matrix into smaller chunks then distributing the computation on multiple nodes. The partial calculations need to be exchanged among the nodes to obtain the final result, and therefore the penalty of the increased network traffic will eventually overrun the benefit of parallelism. Further investigation reveals that the percent of time used in transmitting data increases from 10.5% to 37.2% (from 5 CPUs to 20 CPUs). On the other hand, indexing phase scales very well by using more computation nodes because PANNS does not require exchanging too much data among the nodes.

5.2 Accuracy and Scalability of Searching

Service time represents the amount of time needed to process a request, which can also be used to calculate server throughput. Throughput is arguably the most important metric to measure the scalability of a service. We tested the service time of KServer by using one of the two web servers in the aforementioned testbed. We model the con-

(c,t)		(20,16)	(20,32)	(20,64)	(20,128)	(20,256)	(80,16)	(80,32)	(80,64)	(80,128)	(80,256)
Index (MB)		361	721	1445	2891	5782	258	519	1039	2078	4155
Precision (%)		68.5	75.2	84.7	89.4	94.9	71.3	83.6	91.2	95.6	99.2
$\alpha_1 = 1.0$	ms	2.2	3.7	5.1	5.9	6.8	4.6	7.9	11.2	13.7	16.1
$\alpha_2 = 0.9$	ms	3.4	4.3	6.0	6.8	7.6	7.2	9.5	14.9	15.3	17.1
$\alpha_3 = 0.8$	ms	4.3	4.9	6.7	7.9	8.4	9.1	11.7	15.2	17.4	17.9
$\alpha_4 = 0.7$	ms	5.5	6.3	7.4	8.5	9.3	11.6	13.4	16.1	17.7	18.5
$\alpha_5 = 0.6$	ms	6.1	6.7	7.9	8.8	9.8	13.9	16.0	18.5	19.8	21.1
$\alpha_6 = 0.5$	ms	6.7	7.3	8.2	9.0	10.3	16.6	17.8	19.9	20.4	23.1

Table 3: Scalability test on KServer with different index configurations and request patterns. (c,t) in the first row, c represents the maximum cluster size, and t represents the number of RP-trees. Zipf-(α, n) is used to model the content popularity.

tent popularity with a Zipf distribution, whose probability mass function is $f(x) = \frac{1}{x^\alpha \sum_{i=1}^{n} i^{-\alpha}}$, where x is the item index, n is the total number of items in the database, and α controls the skewness of the distribution. Smaller values of α lead to more uniform distributions while large α values assign more mass to elements with small i. It has been empirically demonstrated that in real-world data following a power-law, the α values typically range between 0.9 and 1.0 [9, 10]. We plug in different α to generate the request stream. The next request is sent out as soon as the results of the previous one is successfully received. Round trip time (RTT) depends on network conditions and is irrelevant to the efficiency of the backend, hence is excluded from total service time. Table 3 summarizes our results.

We also experiment with various index configurations to understand how index impacts the server performance. The index is configured with two parameters: the maximum cluster size c and the number of search trees t. Note c determines how many random vectors we will draw for each search tree, which further impacts the search precision. The first row in Table 3 lists all the configurations. In general, for a realistic $\alpha = 0.9$ and index $(20, 256)$, the throughput can reach 1052 requests per second (i.e., $\frac{1000}{7.6} \times 8$) on a node of 8 CPUs.

From Table 3, we can see that including more RP-trees improves the search accuracy but also increases the index size. Since we only store the random seed for all random vectors which is practically negligible, the growth of index size is due to storing the tree structures. The time overhead of searching also grows sublinearly with more trees. However, since searching in different trees are independent and can be easily parallelized, the performance can be further improved by using more CPUs. Given a specific index configuration, the service time increases as α decreases, which attests our arguments in Section 4.3. Namely, we can exploit the highly skewed popularity distribution to scale up the system. As we mentioned, increasing cluster size is equivalent to reducing the number of random projections, and vice versa. We increase the maximum cluster size from 20 to 80 and present the result in the right half of Table 3. Though the intuition is that the precision should deteriorate with less random projections, we notice that the precision is improved instead of degrading. The reason is two-fold: first, large cluster size reduces the probability of misclassification for those projections close to the split point. Second, since we perform linear search within a cluster, larger cluster enables us explore more neighbors which leads to higher probability to find actual nearest ones. Nonetheless, also due to the linear search in larger clusters, the gain in the accuracy is at the price of inflated searching time.

6. DISCUSSIONS & FUTURE WORK

We mainly focused on the implementation issues in the present paper. It is worth noting that we can improve the current design in many ways. E.g., new content keeps flowing into Kvasir. However, we need not necessarily to rebuild the LSA model and index from scratch whenever new documents arrive. LSA space can be adjusted on the fly for incremental updates [7]. Then, the trees are updated by adding the new points to the corresponding leaf cell.

The results from KServer are ranked based on cosine similarity at the moment. However, finer-grained and more personalized re-ranking can be implemented by taking users' both long-term and short-term preferences into account. Such functionality can be achieved by extending one-class SVM or utilizing other techniques like reinforcement learning [15]. As we also mentioned, the frontend is not constrained within a browser and can be implemented in various ways on different platforms. Kvasir provides a scalable Internet service via a RESTful API. Content providers can integrate Kvasir service on their website to enhance users' experience by automatically providing similar articles on the same page. Besides, more optimizations can be done at frontend side via caching and compression to reduce the traffic overhead.

Currently, Kvasir does not provide full-fledged security and privacy support. For security, malicious users may launch DDoS attacks by submitting a huge amount of random requests quickly. Though limiting the request rate can mitigate such attacks to some extent, DDoS attacks are difficult to defend against in general. For privacy, Kvasir needs tracking a user's browsing history to provide personalized results. However, a user may not want to store such private information on the server. Finer-grained policy is needed to provide flexible privacy configuration. Security and privacy definitely deserve more thorough investigations in our future work. Besides, we are also planning to perform extensive user-studies to evaluate the usefulness of the system.

7. CONCLUSION

In this paper, we presented Kvasir which provides seamless integration of LSA-based content provision into web browsing. We proposed a parallel RP-tree algorithm and implemented stochastic SVD on Spark to tackle the scalability challenges. The proposed solutions were evaluated on the testbed and scaled well on multiple CPUs. The key components of Kvasir are implemented as an Apache Spark library, and the source code is publicly accessible on Github.

8. REFERENCES

[1] Kvasir project, http://cs.helsinki.fi/u/lxwang/kvasir.

[2] Scalanlp, http://www.scalanlp.org/.

[3] scikit-learn toolkit, http://scikit-learn.org/.

[4] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS '06. 47th Annual IEEE Symposium on*, pages 459–468, Oct 2006.

[5] T. Berners-Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[6] M. Berry, S. Dumais, and G. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

[7] M. Brand. Fast low-rank modifications of the thin singular value decomposition. *Linear Algebra and its Applications*, 415(1):20 – 30, 2006. Special Issue on Large Scale Linear and Nonlinear Eigenvalue Problems.

[8] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, pages 43–52, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[9] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134 vol.1, Mar 1999.

[10] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 1–14, New York, NY, USA, 2007. ACM.

[11] P. Cremonesi, A. Tripodi, and R. Turrin. Cross-domain recommender systems. In *Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on*, pages 496–503, Dec 2011.

[12] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 537–546, New York, NY, USA, 2008. ACM.

[13] S. Dasgupta and K. Sinha. Randomized partition trees for exact nearest neighbor search. *CoRR*, abs/1302.1948, 2013.

[14] W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[15] D. Glowacka, T. Ruotsalo, K. Konyushkova, K. Athukorala, S. Kaski, and G. Jacucci. Scinet: A system for browsing scientific literature through keyword manipulation. In *Proceedings of the Companion Publication of the 2013 International Conference on Intelligent User Interfaces Companion*, IUI '13 Companion, pages 61–62, New York, NY, USA, 2013. ACM.

[16] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, Dec. 1992.

[17] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1312–1317. AAAI Press, 2011.

[18] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.*, 53(2):217–288, May 2011.

[19] J. He, W. Liu, and S.-F. Chang. Scalable similarity search with optimized kernel hashing. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 1129–1138, New York, NY, USA, 2010. ACM.

[20] D. Huynh, S. Mazzocchi, and D. Karger. Piggy bank: Experience the semantic web inside your web browser. In Y. Gil, E. Motta, V. Benjamins, and M. Musen, editors, *The Semantic Web - ISWC 2005*, volume 3729 of *Lecture Notes in Computer Science*, pages 413–430. Springer Berlin Heidelberg, 2005.

[21] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(1):117–128, Jan 2011.

[22] Y. Jia, J. Wang, G. Zeng, H. Zha, and X.-S. Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3392–3399, June 2010.

[23] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, Sept. 1999.

[24] Y. Koren and R. Bell. Advances in collaborative filtering. In F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, editors, *Recommender Systems Handbook*, pages 145–186. Springer US, 2011.

[25] B. Li, Q. Yang, and X. Xue. Can movies and books collaborate?: Cross-domain collaborative filtering for sparsity reduction. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 2052–2057, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[26] T. Liu, A. W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. pages 825–832. MIT Press, 2004.

[27] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[28] M. Muja and D. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(11):2227–2240, Nov 2014.

[29] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[30] L. Page, S. Brin, R. Motwani, and T. Winograd. The

pagerank citation ranking: Bringing order to the web. 1999.

[31] R. Rehurek and P. Sojka. Software framework for topic modelling with large corpora. In *Proceedings of LREC 2010 workshop New Challenges for NLP Frameworks*, pages 46–50, Valletta, Malta, 2010. University of Malta.

[32] R. Sproull. Refinements to nearest-neighbor searching ink-dimensional trees. *Algorithmica*, 6(1-6):579–589, 1991.

[33] S. Tasoulis, L. Cheng, N. Valimaki, N. J. Croucher, S. R. Harris, W. P. Hanage, T. Roos, and J. Corander. Random projection based clustering for population genomics. In *Proceedings of IEEE International Conference on Big Data (Big Data), 2014*, pages 675–682. IEEE, 2014.

[34] J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for scalable image retrieval. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3424–3431, June 2010.

[35] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1106–1113, June 2012.

[36] H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu. Complementary hashing for approximate nearest neighbor search. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1631–1638, Nov 2011.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.