Published in final edited form as:

Proc Int Symp High Perform Distrib Comput. 2011 November ; 2011: . doi:10.1145/1996130.1996174.

A Distributed Look-up Architecture for Text Mining Applications using MapReduce

Atila Soner Balkir,

Department of Computer Science, University of Chicago

Ian Foster, and

Department of Computer Science, University of Chicago, Argonne National Laboratory

Andrey Rzhetsky

Computation Institute, University of Chicago, Argonne National Laboratory

Atila Soner Balkir: soner@uchicago.edu; Ian Foster: foster@anl.gov; Andrey Rzhetsky: arzhetsk@medicine.bsd.uchicago.edu

Abstract

Text mining applications typically involve statistical models that require accessing and updating model parameters in an iterative fashion. With the growing size of the data, such models become extremely parameter rich, and naive parallel implementations fail to address the scalability problem of maintaining a distributed look-up table that maps model parameters to their values.

We evaluate several existing alternatives to provide coordination among worker nodes in Hadoop [11] clusters, and suggest a new multi-layered look-up architecture that is specifically optimized for certain problem domains. Our solution exploits the power-law distribution characteristics of the phrase or n-gram counts in large corpora while utilizing a Bloom Filter [2], in-memory cache, and an HBase [12] cluster at varying levels of abstraction.

I. Introduction

Deriving high-quality information from text generally involves working with large data sets, and requires the use of natural language processing methodologies along with statistical models for parameter estimation. These models involve optimization algorithms that perform multiple iterations over the same data set, with each iteration accessing parameter values generated in a preceding iteration until a convergence criterion is met. Thus, we face the problem of providing high-speed access to potentially large members of frequently changing parameter values.

MapReduce [5] has gained significant attention from the scientific community due to its simple yet general programming style. Unlike traditional fine grained parallel programming frameworks such as MPI [6], MapReduce does not allow direct communication between processes, and there is no notion of a shared memory that is globally accessible from all compute nodes. Thus, common information required for the computation is usually distributed to all nodes and loaded into memory during the initialization step of the map or reduce tasks. This approach has been applied in MapReduce implementations of the word alignment problem in phrase-based statistical machine translation [7,8].

This approach of replicating data is clearly limited by the amount of information that can fit in the memory of an individual cluster node. Alternatively, one can store parameter values in a separate relational database. However, applications typically need to access parameter values extremely rapidly. Relational databases are often designed to support more complex data analyse using a query language at the cost of higher latencies compared to a simple distributed hash table that stores only $\langle key, value \rangle$ pairs.

A more feasible solution is to use large scale distributed databases such as HBase, an open source system modelled after BigTable [9]. Although designed to scale well with data size, our experiments show that the current version of HBase (0.20.x) cannot provide the desired low latency since each database request often requires one or more disk accesses.

The disk access problem may partially be solved by replacing HBase with a distributed in-memory database. It is important to emphasize that this approach is different than duplicating and storing the common data in all compute nodes. Information is not duplicated, but divided among the nodes so more data can be stored in a distributed manner. Since the requests are directly served from memory, latencies are generally lower compared to HBase. Memcached [10] is an open source distributed cache originally designed to speed up web applications by reducing the number of requests to a server. It can also be used as an in-memory database if automatic eviction of cached items is disabled, which may be specified as an option to Memcached daemons at start up.

While using Memcached seems to serve the needs of coordination between the compute nodes, there are two drawbacks with this approach. MapReduce is designed for large tasks spanning hundreds to thousands of often low-cost computers running in parallel, making fault tolerance an essential feature. The data on the distributed file system used by MapReduce is replicated on multiple machines, and when a compute node fails during execution, the failed task is assigned to a different node. Originally designed as a caching mechanism, Memcached does not provide any means of fault tolerance. Consequently, if a Memcached node fails during a MapReduce task, the model parameters on that node will no longer be available and the job will fail. Secondly, no matter how fast database requests are served from memory, each distributed table query has to be sent and received over the network, bounding the minimum latency by the network round trip time.

Our contributions in this paper are summarized as follows:

1. We offering a new multi-layered look-up architecture optimized for statistical parameter estimation models where the data follows a power law [1] distribution, having a long tail.
2. We introduce a statistical method to divide sentences in a large corpus into smaller and meaningful phrases by estimating phrase probabilities.
3. We evaluate and compare various implementations of the above model using HBase, Memcached, and our multi-layered solution in terms of end-to-end run time, average latency per request and scalability.

II. Background

- A. MapReduce
- B. Big Table
- C. Memcached
- D. Zipf, Power law, and The Long Tail

III. Sentence Chunking Problem

In the context of this paper, a phrase refers to a collocation or cohesion of consecutive words in text. Discovering recurrent phrases automatically from text has been an interesting research topic due to its wide area of applications. A collocation is defined as a recurrent combination of words that occur more often than chance and that correspond to arbitrary word usages [3]. A *key phrase* is a phrase that occurs multiple times, and can be useful for summarization, topic search or clustering. Several studies have been done previously to address the problem of discovering a subset of words in a given text that best render its topic [4].

The sentence chunking problem is established upon this rich tradition of key phrase detection from free text, but it is different by the objective of discovering all phrases in a large corpus based on the assumption that the same words (unigrams) within the same n-gram may constitute one phrase in some sentences and distinct phrases in others. On the basis of this requirement, a statistical chunker that can infer the most likely way to split every sentence into phrases, taking into account all sentences in the corpus jointly, can be developed.

For a sentence of n words, there are $n-1$ word boundaries and we have the binary decision of chunking or not chunking at each word boundary, thus, there are 2^{n-1} valid chunkings.

Also for each such sentence, a phrase graph containing $\frac{n(n+1)}{2} + 2 = \Theta(n^2)$ nodes has to be generated, since the model requires taking into account all possible phrases. This problem is visualized in Figure 1 as a phrase graph for a sample sentence with four words. Observe that there are 10 nodes including the first and the last one. Any valid chunking of this sentence can be represented as a directed walk from the nodes *Begin* to *End*. The sentence chunking problem is to find the most likely directed walk in the phrase graph, which can be any of the eight alternative routes in this example.

The model assigns a probability to all phrases to compute the likelihood of each route, where a route represents a chunking alternative for the sentence. These probabilities are not known in advance, but can be estimated for the whole corpus. Once the probabilities are estimated, they are used to process the input sentences independently to compute the likelihood of each route. For each sentence, the route having the maximum likelihood is selected as the solution.

The sum of all probabilities assigned to all phrases in the corpus equals to 1, and each individual probability is a number between 0 and 1.

$$\sum_{j=1}^N f_j = 1, \quad 0 \leq f_j \leq 1 \quad (1)$$

where N is the total number of phrases in the corpus. The likelihood value associated with a single chunking C_i of a sentence S is defined as the product of the probability of all of its constituent phrases and a normalization factor z that depends on the number of words in each phrase in C_i .

$$F(C_i | \{f_k\}_{k=1..N}) = z(C_i) \prod_{f_j \in C_i} f_j \quad (2)$$

In a similar manner, the likelihood of a sentence S is given by the sum of the likelihoods of all possible chunking alternatives for S .

$$F(S | \{f_k\}_{k=1..N}) = \sum_{i=1}^{2^n-1} F(C_i | \{f_k\}_{k=1..N}) \quad (3)$$

where n is the number of words in S . Finally, the likelihood of a corpus is computed by multiplying the likelihoods of all of its constituent sentences S_i .

$$F(Corpus = S \{i\}_{i=1..M} | \{f_k\}_{k=1..N}) = \prod_{i=1}^M F(S_i | \{f_k\}_{k=1..N}) \quad (4)$$

where M is the total number of sentences in the corpus. We then take the maximum likelihood estimates for parameters Θ that correspond to the probability of each phrase $\{f_k\}_{k=1..N}$.

$$\Theta_{ML} = \arg \max_{\Theta} F(Corpus | \Theta) \quad (5)$$

For example, given four short sentences (A B C) (A B) (A B D) (D), we have the optimization problem:

$$\begin{aligned} & \textbf{maximize} \\ & [z(A, BC)f(A)f(BC) + z(A, B, C)f(A)f(B)f(C) + \\ & \quad z(AB, C)f(AB)f(C) + z(ABC)f(ABC)] * \\ & \quad [z(A, B)f(A)f(B) + z(AB)f(AB)] * \\ & [z(A, BD)f(A)f(BD) + z(A, B, D)f(A)f(B)f(D) + \\ & \quad z(AB, D)f(AB)f(D) + z(ABD)f(ABD)] * \\ & \quad [z(D)f(D)], \end{aligned}$$

subject to Equation (1).

In practice, these multiplications are represented as sums of logarithms to handle floating point overflows. From these parameters, the optimal chunking of a sentence is determined by computing the maximum weighted path from the *Begin* to *End* nodes in the phrase graph using a dynamic forward algorithm having linear time and space complexity requirements with the number of nodes in the graph. The idea is to accumulate the partial products from the first to the last node as we move forward in the phrase graph. This method is also used for computing the sentence likelihoods in Equation (3) for fast processing.

IV. Implementation

A. Overview of the Algorithms

The sentence chunking application is represented as a Hill Climbing [13] algorithm in Algorithm 1. At line 1, an initial random probability distribution is generated and assigned as the best solution so far. At line 4, an alternative probability distribution is generated by modifying *best[]*. The inner for loop then calculates the likelihood of each sentence sequentially and updates the corpus likelihood regardless of the order of the sentences. Finally, the new corpus likelihood is compared against the maximum likelihood obtained so far, and if a better solution is found, *max* and *best[]* are updated accordingly.

The independent nature of the algorithm allows for easy parallelization by processing sentences concurrently in multiple machines and finally computing the corpus likelihood by an aggregation step. Lines (6–8) can be described as a MapReduce algorithm in Algorithm 2

At the first line, a look-up table reference to a storage system is obtained. The notion of a storage system in this context is highly abstract and it can be anything from a local hash table kept in memory to a distributed database table in a cluster. The fundamental functionality of this table is to map model parameters (phrases) to their corresponding values (probabilities) efficiently. We examine several alternative storage systems in Section V.

The values received as input at the map stage are sentences from the input corpus. A pre-processing step converts all letters to lower case and removes the punctuation mark at the end of the sentence. Then the sentence likelihood is computed at line 5 by generating the phrase graph of the input sentence and applying Equation (3). The resulting value is then emitted as intermediate output from the mapper with a constant *byte* key, set to 1. There is only once reducer process which collects all sentence likelihoods from the mappers and simply multiplies them, yielding the final corpus likelihood. Although the reduce step is a serial operation, it is typically fast because combiners running after the map tasks are used to perform most multiplications locally, decreasing the size of the data going through the pipeline and thus the number of input records to the reducer.

B. Scalability Challenges

Note that the mapper processes must query the look-up table constantly during the computation. Since the model must generate all phrases in the corpus to estimate their probabilities, the above naive implementation needs to access the probability of each unique phrase one or more times, proportionally to its frequency in the corpus. For example, if a phrase occurs *j* times in the corpus, it will be generated inside a phrase graph *j* times,

possibly in different machines being a part of different sentences. This requirement has two immediate effects on performance:

- All phrases in the corpus must be generated and stored along with their current (most up-to-date) probability estimates in the form of $\langle key, value \rangle$ pairs in a lookup table. The storage requirements of such a table grow rapidly with increasing corpus size.
- The mappers require fast access to these parameters since an average English sentence containing 20 words requires hundreds of look-ups to the storage system. Small changes in the average look-up latencies per phrase can make big differences in overall application run time.

C. Approximations and Parameter Clustering

The number of parameters to be estimated by Algorithm 1 grows fast. Generating all possible phrases in a few Megabytes of plain text corpus yields millions of phrases. The parameter space becomes several orders of magnitude larger with corpora of considerable size scaling to Gigabytes. This gigantic nature of the problem necessitates some approximations by clustering the parameters using a relevant feature.

We assume phrase probabilities belong to C distinct classes and have one parameter for each class. Phrases are assigned to a specific class based on their frequency in the corpus. For example, all phrases that occur only once in the corpus are grouped in $class_1$. In general, all phrases that occur j times in the corpus belong to $class_j$, where the probabilities of all phrases in $class_j$ are equal.

$$p_k \in class_j \iff freq(p_k)=j \quad (6)$$

Subsequently, Equation (1) can be re-written as:

$$\sum_{j=1}^C f_j |class_j| = 1, \quad 0 \leq f_j \leq 1, \quad C \ll N \quad (7)$$

where $|class_j|$ indicates the cardinality of $class_j$, showing the number of unique phrases that occur j times in the corpus, and f_j is the probability associated with each such phrase.

Clustering phrases into frequency classes reduces the number of variables. However, keeping track of how many times each phrase occurs in the corpus remains to be an application requirement. In order to map a *phrase* to its *probability*, two look-up operations have to be issued, instead of one. First, a table storing $\langle phrase, frequency \rangle$ pairs is queried to retrieve the frequency class of the phrase. Second, another table containing $\langle frequency, probability \rangle$ pairs is queried to match that frequency class with a probability. A comparison of storage requirements for both tables with increasing corpus size is given in Table I.

The first table is large because it contains all unique phrases in the corpus along with their frequencies. It is generated by a MapReduce application, which is a variation of the widely known word-count example [5], and uploaded to a storage system. Given the low look-up

latency requirements while providing fault tolerance and scalability at the same time, it must be stored in a distributed architecture. The second table is much smaller compared to the first one since the total number of rows it contains is limited by C , the number of distinct frequency classes. It can easily fit in a small file not larger than a few Megabytes even considering very large problem instances spanning Gigabytes of text input. At each iteration, compute nodes read its contents from the shared file system and store it in a hash table locally. Essentially, this file always contains the best probability estimates obtained till then. As specified at line 4 of Algorithm 1, it is updated to generate a new distribution by modifying the current one at the beginning of each iteration. The operation of these tables are shown in Figure 2.

V. Implementation

A. Reducing Data Size

An evident choice for managing large scale data is to distribute them among multiple computers while providing load balance. This is usually done by loading the whole data set to a distributed table, followed by creating an index either manually or automatically depending on the platform. Without knowing anything further about the data, loading stage is obviously mandatory. As discussed previously, the large data set contains $\langle key, value \rangle$ pairs mapping *phrases* to their *frequencies* in the implementation of the sentence chunking model. A simple design would necessitate loading the whole data set to the distributed table, causing storage requirements of considerable size and possibly leading to additional overheads for maintaining it across multiple machines. Compressing or hashing the records will clearly result in some storage gains, but they do not reduce the number of rows in the table.

A good observation on the distribution of phrase frequencies reveals that most phrases occur only once in the corpus, and they are usually longer than repeating phrases. This is not surprising because the model generates all possible phrases in the corpus including whole sentences. Figure 3 shows the difference between the number of unique phrases that occur *twice or more*, and the number of phrases that occur *only once* in the corpus with growing input size. The lines are estimated using ordinary least squares method to emphasize the linear relationship between the number of phrases and the input corpus size. Observe that n_1 is the number of phrases whose frequency equals to *one*, while n_{2+} shows the number of unique phrases whose frequency is at least *two*.

The most immediate consequence of this observation is that it allows to decrease the number of rows in the distributed table. This is accomplished by only considering those phrases whose frequency is greater than 1, and ignoring the rest during the loading stage. When the table is queried to retrieve the frequency of a phrase, the returned result is compared against *null*. A *null* result indicates that the table does not contain a row for that phrase, implying its frequency is equal to 1. This is summarized in Algorithm 3.

B. HBase vs. Memcached

HBase is a good open source candidate for storing phrase frequencies as it utilizes core Hadoop components like HDFS, and provides a rich set of tools to support MapReduce

applications in its API including special table input/output formats for Hadoop programs. Since our model requires multiple iterations over the corpus, data availability is critical. HBase provides fault tolerance and reduces the risk of data loss qualifying as a good fit. However, there is a drawback in look-up latencies at the cost of fault tolerance and high availability due to potentially long disc access times needed to fetch data.

The disc access barrier can be circumvented by storing phrase frequencies entirely in a distributed memory system like Memcached. Technically, using Memcached does not contradict with our initial requirement of having a *scalable* storage system because new nodes can always be added to the cluster as the input size increases. Memcached results in reduced look-up latencies compared to HBase, but there is no fault tolerance so it should only be considered for clusters of smaller size.

In our experiments, we used the same cluster nodes both for running the Hadoop cluster, and the storage system instances.

C. Utilizing a Bloom Filter

A common performance issue in both alternatives above is that each request and response must go over the network. This bounds the minimum look-up latency by the network round trip time (RTT). Remember that most phrases have a frequency equal to 1. When a request is issued to the storage system (HBase or Memcached), it goes over the network and most of the time a *null* reference is returned in response, which is further replaced by 1. For each such query, the application stalls for at least one network RTT.

We use a Bloom Filter [2] to cut down the number of queries to the storage system by a considerable amount. A Bloom Filter is a combination of a bit vector and k deterministic hash functions. Figure 5.3 shows an example Bloom Filter with two hash functions containing three records.

1) Inserting a Phrase—When inserting a phrase into the filter, k bit locations are marked in the vector by computing $hash_k(phrase) \% n$, where n is the total number of bits. In the sentence chunking application, a Bloom Filter for an input corpus is constructed in parallel with a single MapReduce program. The input is all $\langle phrase, frequency \rangle$ pairs with $frequency > 1$. At the map stage, each compute node starts with an empty bit vector, and inserts its own share of input phrases locally. Then, vectors coming from all mappers are grouped together, and a *bitwise OR* operation is applied to produce the final output at the reduce stage.

2) Querying a Phrase—The size of a Bloom filter can be customized according to the error rate. It is possible to construct Bloom Filters of small size with an error rate of 1% even for large problem instances. Compute nodes store a copy of the Bloom Filter in local memory for fast access. To fetch the frequency of a phrase from the storage system, a compute node first queries its local copy of the Bloom Filter by checking the corresponding bit locations. If any of the k bits is not set, there is no need to issue a further storage system query, and it immediately follows that the frequency of the phrase equals to 1. A false positive happens when all k bit locations are set in the Bloom Filter, but the storage system

does not contain an entry for the input phrase. In this case, a *null* reference is returned from the storage system, and replaced by 1. In all other cases, a *frequency* > 1 is returned. The querying process is summarized in Algorithm 4.

D. Caching Results

It is possible to further decrease the look-up latencies by efficiently caching the results returned from the storage system. Figure 4 shows the cardinality of the frequency classes plotted on logarithmic scale.

The linear plot with the long tail towards the end indicates that the data follow a power law distribution. There are a large number of phrases with low frequency, and a small number of phrases with very high frequency in the corpus. We use local memory to cache the records returned from the storage system. The caching policy is LRU, which gives a good approximation of the phrase frequencies, increasing cache utility. Common phrases in the corpus are accessed more frequently, tending to stay longer in the cache. When retrieving a phrase frequency, the Bloom Filter is queried first for membership test. If it returns a *TRUE* answer, the local cache is checked. If the local cache contains an entry for the phrase, it is immediately returned eliminating the need for a remote look-up. Otherwise, the storage system is queried, and if the frequency of the phrase is greater than one, it is inserted in the cache for future reference. A state diagram describing the operation of the entire look-up architecture is shown in Figure 5.7.

E. Limitations of Hadoop

Hadoop creates a separate Java Virtual Machine (JVM) for each map or reduce task to isolate them from others running on the same node by default. Once a task is over, its JVM is destroyed. This design choice is consistent with the basic assumption of the MapReduce model where isomorphic tasks do not need to communicate, and the only communication happens during the *shuffle* stage. In some cases, tasks may need to share common data, usually in the form of *static* objects. For example, in our application all mappers need access to a Bloom Filter, and a local cache stored in the memory of each compute node. However, *static* objects are local to the JVM they are created in, so it is not possible to share them among multiple mappers by default.

This complication is resolved by running mappers on the same node as threads in a single JVM, rather than separate processes on multiple JVM's. In each node, there is only one JVM that schedules tasks and maintains a thread pool. This is accomplished by setting the maximum number of map tasks per JVM to infinity, enabling *JVM reuse*, and using the *MultithreadedMapper* class in Hadoop. The static objects are defined as *singletons* at the beginning, and the JVM is only destroyed if there are no more tasks to run in that node.

VI. Experiments

All experiments were conducted on a five node Hadoop cluster with one node being the task tracker and the name node, and the rest four nodes being workers. Each node has two Intel Xeon 2.80 GHz CPU cores, and 2 GB of memory. We used Hadoop 0.20.1 and HBase 0.20.0 along with Java 1.6.0 13. Each worker node also ran Memcached 1.2.2 servers with

128 MB of pre-allocated RAM space when necessary. For each experiment, we performed five runs and reported the arithmetic mean.

A. Running Time

The first experiment measures the run time performance of the sentence chunking application for *one* iteration on six different storage mechanisms. We intentionally performed this experiment with a small input corpus of size 120 MB to compare the performance of an in-memory hash table against other distributed alternatives for storing phrase frequencies.

Figure 5 shows the results. We obtained the lowest running time with the in-memory hash table because there are no performance issues like network latencies or scanning database files. This experiment also demonstrates that the main performance bottleneck is the network latency, not the overhead that comes from searching database files to retrieve records. Observe that Memcached is only twice as fast as HBase, although it stores everything in the memory. An in-memory hash table achieves 23 times better performance compared to Memcached, and the principle difference between these two systems is serving the queries locally vs. remotely over the network.

B. Average Latency

We measured the average latency of retrieving a phrase frequency by computing the time difference between requesting and receiving a record. Each mapper emits its total latency and the number of read requests as a $\langle \text{key}, \text{value} \rangle$ pair, and a reducer aggregates the results, finally dividing the overall latency by the total number of read requests. The results are shown in Figure 6.

The in-memory hash table achieves the lowest latency which is approximately 10^{-3} milliseconds, and thus not visible in this figure. The addition of a Bloom Filter noticeably lowers the latency of HBase and Memcached look-ups. Furthermore, the combination of *HBase + Bloom Filter + local cache* performs better than *Memcached + Bloom Filter*. This implies that utilizing the physical memory of the compute nodes locally is more efficient than making use of them as part of Memcached.

C. Scalability

The first two experiments demonstrate that *HBase + Bloom Filter + local cache* combination yields lower run time and latency compared to the other distributed alternatives while providing fault tolerance at the same time. Nevertheless, an important characteristic of a storage system is its sensitivity to changing input size. Ideally, increasing the corpus size should not raise the average latency of requesting a phrase frequency.

In Figure 3 it is shown that n_1 and n_{2+} grow linearly with input (N), where they correspond to the number of unique phrases having frequency equal to *one*, and at least *two* respectively. We can write n_{2+} as a linear function of n_1 as $n_{2+} = an_1 + b$ easily by solving for N in both equations. Furthermore, Section VI-B revealed that the average time to retrieve a phrase frequency from the memory is negligible compared to fetching it from HBase.

Let t be the time to retrieve a phrase frequency from HBase, and p be the false positive ratio of the Bloom Filter. The expected number of requests to HBase is given by $n_1p + n_{2+}$. If we assume a worst case behavior by ignoring the local cache, the average latency becomes:

$$\frac{(n_1p+n_{2+})t}{n_1+n_{2+}} = \frac{(n_1p+an_1+b)t}{n_1+an_1+b} \approx \frac{(p+a)t}{1+a} \quad (8)$$

for large n_1 . This implies that the average latency of the storage system stays constant if p and t do not get larger with growing input size. Maintaining a fixed false positive probability is possible by increasing m , length of the bit vector linearly with the number records inserted [2]. The optimal length of the Bloom Filter for a given p is computed by:

$$m = \frac{-pn_{2+}}{[\ln(2)]^2} \quad (9)$$

Figure 7 shows the average latency of the storage system with increasing input.

For each point in the graph, the size of the Bloom Filter is computed by linearly estimating n_{2+} from the corpus size N , and then plugging it in Equation 9 for $p = 10\%$. The slight increase in latency is because the data load per machine increases in HBase, and t does not stay constant. However, this can be addressed by adding more nodes to the cluster if it becomes a bottleneck.

VII. Conclusion

The conclusion goes here.

Acknowledgments

The authors would like to thank...

References

1. Newman MEJ. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*. 2005; 46:323–351.
2. Bloom, BH. Space/time trade-offs in hash coding with allowable errors; *Communications of the ACM*. 1970. p. 422-426.
3. Ahonen-Myka H, Doucet A. Data Mining Meets Collocations Discovery. *Inquiries into Words, Constraints and Contexts, Festschrift for Kimmo Koskeniemi (CSLI Studies in Computational Linguistics)*. 2005:194–203.
4. Turney PD. Learning Algorithms for Keyphrase Extraction. *Information Retrieval*. 2000:303–336.
5. Ghemawat, S.; Dean, J. MapReduce: Simplified Data Processing on Large Clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*; 2004.
6. Message Passing Interface Forum. MPI: A message passing interface. *Proceedings of the IEEE Supercomputing*; 1993. p. 878-883.
7. Brown PF, Della Pietra VJ, Della Pietra SA, Mercer RL. The Mathematics of Statistical Machine Translation: Parameter Estimation. *Computational Linguistics*. 1993:263–311.
8. Dyer, C.; Cordova, A.; Mont, A.; Lin, J. Fast, easy, and cheap: construction of statistical machine translation models with MapReduce. *Proceedings of the Third Workshop on Statistical Machine Translation*; 2008. p. 199-207.

9. Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, WC.; Wallach, DA.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, RE. Bigtable: A Distributed Storage System for Structured Data. OSDI'06: Seventh Symposium on Operating System Design and Implementation; 2007.
10. Memcached: A Distributed Memory Object Caching System. <http://memcached.org/>, [Online]
11. Apache Hadoop. <http://hadoop.apache.org/>, [Online]
12. Apache HBase. <http://hadoop.apache.org/hbase/>, [Online]
13. Russell, SJ.; Norvig, P. Artificial Intelligence: A Modern Approach. 2. Upper Saddle River, New Jersey: Prentice Hall; 2003.

Fig. 1.
Phrase Graph

Fig. 2.
Storage Tables in Operation

Fig. 3.
Phrase Frequency Distribution

Fig. 4.
Long tale distribution of the phrase frequencies

Fig. 5.
End-to-end running time per one iteration

Fig. 6.
Average latency for a single look-up

Fig. 7.
Average look-up latency with growing input size

TABLE I

Comparison of Storage Requirements

Input	phrase $\rightarrow$ freq		freq $\rightarrow$ prob	
	Rows	Size	Rows	Size
15 MB	26×10^6	2.7 GB	989	28 KB
30 MB	53×10^6	5.6 GB	1470	41 KB
60 MB	110×10^6	11.2 GB	2155	60 KB
90 MB	153×10^6	16.4 GB	2650	74 KB
120 MB	206×10^6	21.6 GB	3100	87 KB

Algorithm 1**Sentence Chunker with Hill Climbing**

```

1:   $best[] \leftarrow$  generate random distribution
2:   $max \leftarrow -\infty$ 
3:  for  $j = 1$  to  $MAX\_ITERATIONS$  do
4:     $f\{k\}_{k=1..N} \leftarrow$  alternate( $best[]$ )
5:     $product \leftarrow 1$ 
6:    for  $i = 1$  to  $M$  do
7:       $prod \leftarrow prod * F(Sentence_i | f\{k\}_{k=1..N})$ 
8:    end for
9:    if  $prod > max$  then
10:      $max \leftarrow prod$ 
11:      $best[] \leftarrow f\{k\}_{k=1..N}$ 
12:    end if
13:  end for
14:  return  $best$ 

```

Algorithm 2**Sentence Chunker MapReduce**

```

1:  table  $\leftarrow$  obtain a storage system reference
2:  map(Key k, Value S)
3:  begin
4:    S  $\leftarrow$  preProcess(S)
5:    val  $\leftarrow$  computeLhood(S,table) //equation (3)
6:    emit(1, val) //constant key
7:  end
8:  reduce(Key k, Value Lhoods[])
9:  begin
10:   product  $\leftarrow$  1
11:   foreach value v in Lhoods[]
12:     product  $\leftarrow$  product * v
13:   end for
14:   emit(k, product)
15: end

```

Algorithm 3

Retrieve Phrase Frequency

```
1:  getFrequency(table t, phrase p)
2:  begin
3:    result  $\leftarrow$  t.query(p)
4:    if result = null then
5:      result  $\leftarrow$  1
6:    end if
7:    return result
8:  end
```

Algorithm 4

Retrieve Phrase Frequency with Bloom Filter

```

1:  getFrequency(table t, bloomFilter b, phrase p)
2:  begin
3:    flag  $\leftarrow$  b.contains(p)
4:    if flag = FALSE then
5:      result  $\leftarrow$  1
6:    else
7:      result  $\leftarrow$  t.query(p)
8:      if result = null then    //false positive
9:        result  $\leftarrow$  1
10:     end if
11:   end if
12:   return result
13: end

```