Statistical Sentence Chunking Using Map Reduce

Atilla Soner Balkir
Department of Computer Science
University of Chicago

Winter 2010
Table of Contents

Abstract ................................................................................................................................................... 4

Introduction ............................................................................................................................................. 5
1.1 Expinions ....................................................................................................................................... 5
1.2 Phrase Discovery ........................................................................................................................... 6
1.3 Map Reduce................................................................................................................................... 8
1.4 Distributed Database and Cache architectures ............................................................................. 9
1.4.1 BigTable .................................................................................................................................. 9
1.4.2 Distributed Caching and Memcached .................................................................................. 12
1.5 Our Methods and Results ............................................................................................................ 13

Background ............................................................................................................................................ 15
2.1 Theory .......................................................................................................................................... 15
2.1.1 Word Segmentation Problem ............................................................................................... 15
2.1.2 Sentence Chunking has Similarities to Word Segmentation ................................................ 16
2.1.3 Minimum Description Length Analysis ................................................................................. 17
2.1.4 Key Phrase Discovery ............................................................................................................ 17
2.1.4 Collocations Discovery .......................................................................................................... 19
2.2 Systems ........................................................................................................................................ 21
2.2.1 MapReduce in Scientific Applications .................................................................................. 21
2.2.2 Machine Learning ................................................................................................................. 21
2.2.3 Statistical Machine Translation ............................................................................................ 22
2.2.4 Graph Search ........................................................................................................................ 24
2.2.5 Genetic Algorithms ............................................................................................................... 25

Design and Implementation .................................................................................................................. 27
3.1 Chunking Models ......................................................................................................................... 27
3.1.1 Statistical Model for Sentence Chunking ............................................................................. 27
3.1.2 Heuristic Rule Based Model for Sentence Chunking ............................................................ 31
3.2 Implementation ........................................................................................................................... 33
3.2.1 Counting Phrases .................................................................................................................. 33
3.2.2 Creating a histogram of phrase counts ................................................................................ 34
3.2.3 Introducing Problems of Scale............................................................................................. 36
3.2.4 Filtering phrases with count > 1 ........................................................................................... 36
3.2.5 Initial implementation with HBase ....................................................................................... 37
3.2.6 Uploading Phrase Counts to a Distributed Storage System ................................................. 38
Abstract

Extracting concepts and relations within statements across a large corpus requires multiple steps of related processes: chunking a sentence into its constituent terms (phrases), semantically classifying them, and creating new statements by assembling those phrases using knowledge extraction. We focus on the first step of this series of processes by the design and evaluation of two novel algorithms for chunking sentences using probabilistic measures. One important aspect of this problem is the requirement for parallel computation methods given our goal of scaling up to millions of sentences and billions of words. A key technical component of the implementation is the use of Map Reduce for processing large data sets and Big Table for managing distributed data while providing random, real time read/write access.
Chapter 1

Introduction

1.1 Expinions

Science is characterized by myriad facts. These facts may appear simple individually. Achieving a broader understanding of these facts by synthesizing them, however, is an overwhelming challenge. We embark a series of formidable tasks when we want to study a specific scientific topic. These are:

(a) Determining what is relevant to the topic
(b) Finding out where to look at for information
(c) Identifying relevant scientific results across volumes of unrelated documents in the existing literature
(d) Putting these noisy, various statements into a more coherent form
(e) Discovering novel connections between seemingly unrelated phenomena and generating new hypotheses

This work is part of a larger project on developing methods that will allow for the automatic large-scale harvesting, analysis, mapping and sharing of expert opinions, experiences, and even speculation, which we refer as expinions.

Many uncertainties are presented when scientist want to model complex systems like human brain, sociological communities and global market economy, making reasoning more complicated. Some pieces in these models may be missing, and some pieces may have unknown reliabilities. If expinions could be drawn out and mapped onto existing scientific hypotheses and positive findings, they would then be immediately useful for refining models of the complex system of scientific knowledge. Expinions could directly estimate the probability of methods and statements using expert knowledge, and add missing links to a probabilistic graph of connected statements using negative knowledge and expert hunches.

In order to achieve this, we need to extract information from literature, build probabilistic maps of scientific knowledge using that information, and elicit new expinions to enhance those maps. This presents the need for developing statistical and computational tools to semantically parse documents in a large corpus to extract concepts and relations, which involves three related processes: chunking a sentence into its constituent terms (phrases), semantically classifying them, and creating new statements by assembling those phrases using knowledge extraction.

This thesis focuses on the first step, chunking a sentence into smaller phrases using a statistical model, and implementing it by taking advantage of distributed computing tools to scale the algorithm. Our work is distinct from the existing work done in the field by its
objective of discovering all phrases in a large corpus based on an iterative probabilistic analysis and being language independent.

1.2 Phrase Discovery

In the context of our work, a phrase refers to a collocation or cohesion of consecutive words in text. Discovering recurrent phrases automatically from text has been an interesting area of research for its wide area of usage. Ahonen-Myka and Doucet (1) define a collocation as a recurrent combination of words that occur more often than chance and that correspond to arbitrary word usages. They define the collocations discovery problem as the process of discovering interesting phrases in a given text. A similar term, key phrase is used for phrases that occur multiple times, and can be useful for summarization, topic search or clustering. Turney (2) (3) described a genetic algorithm based system for automatic extraction of key phrases using parameterized heuristic rules. Frank et Al. (4) named a procedure for domain specific key phrase extraction based on naive Bayes learning schemes. All these studies involve the problem of discovering a subset of subsequent words in a given text that best render its topic.

Our purpose in this study is based on this rich tradition of key phrase detection from free text, but it is different in the sense that we want to discover all phrases in a large corpus based on the assumption that that the same words (unigrams) within the same n-gram may constitute one phrase in some sentences and distinct phrases in others. Based on this, we develop a statistical chunker that can infer the most appropriate way to split every sentence into phrases, taking into account all sentences in the corpus jointly.

Consider the sentence “This is an example”. We can chunk this sentence into phrases as This is - an - example. Another valid chunking would be This - is an - example. Indeed, for any sentence that has \( n \) words, there are \( 2^{n-1} \) valid chunkings because there are \( n-1 \) word boundaries and we have the binary decision of chunking or not chunking at each word boundary. For a sample sentence with four words, we can visualize this problem in Figure 1.1.

![Figure 1.1 – Sentence graph for a sentence containing four words](image)

**Figure 1.1 – Sentence graph for a sentence containing four words**: Each directed walk from the “Begin” to the “End” node is an alternative chunking, which grows exponentially with the number of words in the sentence.
We can represent any valid chunking of this sentence as a directed walk from Begin to End. In the above example, the chunking This - is an - example corresponds to the following directed walk: Begin - 1 - 2,3 - 4 - End. The problem is now defined as finding the most appropriate directed walk in this graph. Of course, the term most appropriate is highly subjective and it depends on the measures we define to choose among multiple paths at each node. In the following chapters, we will describe two chunking models to infer the most appropriate way to split sentences into phrases. The first method is based on a statistical model, which requires multiple iterations over the input corpus in order to maximize an objective function. The second one is built on rather simple heuristic rules, which take into account the frequency and count of each phrase in the corpus.

Both of these models need to keep track of the frequency (count) of each phrase in the input. This requires generating all phrases in the corpus, grouping the same phrases together, and calculating the frequencies by applying an aggregation operation. Our sample corpus contains scientific articles in plain text form, and the size of the corpus varies from 10 MB to 200 MB. However, generating every possible phrase in the corpus populates the data and produces a large intermediate output in the form of <phrase, count> pairs. As an example, generating and counting all phrases in the 200 MB sized corpus yields approximately 50 GB of intermediate data. These properties of our models necessitate the adoption of data intensive parallel programming tools.

Ideally, we need to partition the input and distribute it over multiple machines. Each machine can work independently on its own share of input, and generate all possible phrases as an intermediate output. Furthermore, we need to use grouping and aggregation operators to bring same phrases together on the same computation node, and finalize the counting operation. This involves moving data over the network along with communication and coordination among compute nodes.

The use of parallel computation tools is not only limited to calculating phrase frequencies. The statistical chunking model relies on assigning a random likelihood estimate to each phrase and applying a first choice hill-climbing algorithm to discover a better state than the current one in the search space.

This work involves altering some percentage of the likelihoods at each step, and repeating this process in an iterative fashion until some stopping criterion is met. Reading and updating the likelihood estimates at each iteration requires the need for random read/write access to these values. Additionally, some phrases occur more than once in the corpus and multiple processes running on different machines at the same time may use a likelihood associated with such a phrase. This introduces the need to hold these likelihoods in a shared database, which is ideally distributed and running on the same cluster nodes performing the computation for scalability and performance issues.

We build our parallel processing architecture on top of Apache Hadoop (5), an open source implementation of the Map Reduce (6) programming paradigm introduced by Google. As a distributed storage system, we use HBase (7), a distributed database platform, tightly coupled with Hadoop and HDFS and based on Google’s BigTable (8) architecture. To optimize the performance of the distributed storage system, we further utilize Memcached (9), a low latency, distributed hash table. We discuss these systems in more detail in the forthcoming sections.
1.3 Map Reduce

Parallel applications are typically divided as fine grained or coarse grained depending on the execution time of individual tasks, and the amount of intermediate data generated and transferred between running instances. Fine-grained parallelism is mostly accomplished by message passing such as MPI. Coarse-grained parallel applications are generally described by a workflow, which is an execution graph composed of multiple programs/scripts that have dependency relationships and are related by various trigger relations. Thus, coarse grained parallelism typically requires a more complicated execution engine that is capable of managing large scale science and engineering workflows. Swift (10) and Taverna (11) are examples of such systems. Workflow systems usually come with a run time responsible for managing the data and coordination among the worker nodes, and a programming interface to describe the workflows.

As an alternative to workflow systems, Google introduced a new programming paradigm called Map Reduce. It is a programming abstraction based on the map and reduce primitives in functional programming languages. Their motivation is based on the observation that many parallel applications involve mapping an input to an intermediate data set, partitioning the intermediate data among the compute nodes, and finally applying a reduction function to process groups of records sharing a common property. Several implementations of map reduce have been released, most of which are open source and support a large collection of programming languages to write map or reduce functions. Among the most widely known implementations are Apache Hadoop, Disco (12) and Qizmt (13).

In a map reduce application, the records are represented by <key,value> pairs and the computation starts by invoking one or more mapper processes on each cluster node. Mappers transform each input record to zero or more intermediate output records, initially written to local disk. Once the map operation completes, intermediate records are sorted and grouped together according to their keys. Records sharing the same key are then sent to a particular cluster node, which is typically achieved by applying a hash function to the keys of each group, usually referred as shuffling. After the shuffle operation, groups sharing the same key are merged to a single group and sent to the reducer function. Reducers receive a group of records with the same key, and produce the final output by iterating over the values and performing the necessary operations.

Map Reduce run times usually require a distributed file system to manage the data stored on inexpensive disks. Some examples include GFS, The Google File System (14) and HDFS, Hadoop Distributed File System (5). Large input files are divided into smaller blocks of storage units called splits. Computation takes place on the same cluster nodes holding the data to minimize data transfer and reduce communication costs. This is achieved by trying to schedule a map or reduce task to a worker node, which contains the corresponding input split locally. Replicas of data blocks are created on different nodes, and fault tolerance mechanism is implemented by re-executing a failed task on a different node having the same input block.

Complex workflows can be represented as chains of map reduce applications following one another. This allows the programmer to divide the workflow into smaller pieces and enables easy debugging and profiling. On the flip side, describing large workflows as multiple map
reduce applications may not be trivial and can introduce additional complexities, such as unnecessary disk writes, or sorting of intermediate data although not needed. To address this problem, various higher-level language constructs have been developed, such as Pig (15) and Hive (16). These frameworks give the programmer the ability to express a workflow using a higher-level code, which is then optimized and compiled into a series of map reduce jobs automatically.

Apache’s open source map reduce implementation Hadoop has gained significant attention from the scientific community since its first release in 2006 and various scientific applications have been adapted to the map reduce model in the past few years. Chu et Al. (17) re-modeled a variety of machine learning algorithms including k-Means, Naïve Bayes, Support Vector Machines, Neural Networks and Gaussian Discriminant Analysis using Map Reduce and achieved linear speed-ups in most of them. Kang et Al. (18) described HADI, a fast and highly scalable graph search algorithm for exploring the diameter of web graphs based on Map Reduce. Dyer et Al. (19) designed Map Reduce implementations of two algorithms for parameter estimation in statistical machine translation models. Their approach builds on the use of map reduce for calculating conditional probability distributions by counting joint events and extracting a marginal count for each event in the input set.

We use map reduce for numerous purposes in this project, including generating and counting all the phrases in a large corpus, uploading these phrase counts to a distributed database, filtering phrases that occur more than once in the corpus and performing two different chunking algorithms which involve multiple stages of map reduce operations for several purposes specific to each algorithm. We implement our applications using Hadoop due to its popularity, ease of use, and rich set of Java APIs.

1.4 Distributed Database and Cache architectures

1.4.1 BigTable

BigTable is a high performance distributed storage system designed for scalability and operation on large clusters. It uses a simple data model allowing it to scale and serve low latency read and write requests in petabyte scales.

A row key, a column key and a timestamp are used to identify each cell. Row keys are sorted lexicographically, and maintained in sorted order on disk. This property can be exploited by users to minimize communication and physical I/O. For example, consider a table used to store web page contents. The rows can be stored with the following hierarchical structure:

<table>
<thead>
<tr>
<th>row key</th>
<th>contents:current</th>
<th>time stamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>edu.bio.uchicago/homepage.html</td>
<td>..................</td>
<td>t₂</td>
</tr>
<tr>
<td>edu.cs.uchicago/main.html</td>
<td>..................</td>
<td>t₁</td>
</tr>
<tr>
<td>edu.cs.uchicago/index.html</td>
<td>..................</td>
<td>t₃</td>
</tr>
</tbody>
</table>

Therefore, rows belonging to the same domain are stored close to each other, resulting in efficient analysis with minimum disk reads.

Column keys in a table are identified by a column family and a qualifier. A column family is a group of column keys usually with the same type. In order to add a column key to a table, a
column family must be created. Once the column family is created, any column key can be associated with that family. As an example, suppose we want to store new likelihood estimates of phrases for multiple iterations in a table. We can create a table with only one column family called `iteration` and start inserting values with column keys as `iteration:x` where `x` shows an arbitrary iteration number. The table would then look as follows:

<table>
<thead>
<tr>
<th>phrase</th>
<th>iteration:0</th>
<th>iteration:1</th>
<th>iteration:5</th>
<th>iteration:11</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>5.2E-8</td>
<td>2.4E-6</td>
<td>0.8E-14</td>
<td>-</td>
</tr>
<tr>
<td>going to</td>
<td>9.6E-14</td>
<td>-</td>
<td>2.8E-8</td>
<td>-</td>
</tr>
<tr>
<td>run out of</td>
<td>4.3E-7</td>
<td>-</td>
<td>-</td>
<td>7.4E-21</td>
</tr>
</tbody>
</table>

Notice that we only have to create a column family and not individual columns. New columns belonging to an existing family are created automatically without additional cost, as new write requests arrive. In the above example, there is no new likelihood estimate for the phrase “going to” in the first iteration, so it does not occupy any physical space, not even a null bit. This sparse storage structure is one of the key differences of BigTable that makes it simple yet more scalable and powerful than traditional relational database systems.

Row ranges for each table are partitioned dynamically as new rows are inserted into the table. The basic unit of distribution is called a tablet, which identifies a row range by its last row. Tables are composed of multiple tablets, and each tablet can be stored on any cluster node. Tablets are composed of one or more storage files called SSTable. SSTables are ordered immutable files with an index structure written at the end, and used for locating records faster. In summary, the storage hierarchy in BigTable can be summarized as: `table > tablet > ssfile > record`.

In a BigTable cluster, there is one master node and multiple tablet servers. The master is responsible for assigning tablets to tablet servers, load balancing, and detecting tablet server failures. Clients send read and write requests directly to the tablet servers, without communicating with the master. BigTable uses a distributed locking mechanism called Chubby for a variety of tasks including tablet server discovery, table schema information and bootstrapping the cluster.

There is a three-level hierarchy for locating tablets in the cluster. The first level is a Chubby file that contains the location of the root tablet. The root tablet holds the location of the METADA tablets that belong to a METADATA table in the system. The root tablet is special and never split to guarantee the three-level hierarchy. Finally, the METADATA table contains the location of the user tablets in the database. When a client wants to locate a tablet containing a particular row, it has to issue three round trips, which is not desirable. Therefore, clients cache tablet location information for future access as they query the metadata tables. Figure 1.2 summarizes the three-level hierarchy in BigTable.
Figure 1.2 – Three-level hierarchy for tablet location: Each client performs at most three look-up operations to locate a tablet. Once a tablet is located, this information is cached at the client side for future references to the same location.

Updates to a table are classified into two categories as recent updates and older updates. Older updates are stored in a sequence of SSTables on the file system. Recent updates are inserted into a sorted in memory table called *memtable*. When a write or delete request is issued by a client, it is inserted into the memtable. When a read request arrives at a tablet server, it is executed on both the *memtable* and the SSTables, and the most recent state of the requested record is returned to the client. When the memtable becomes too large due to updates, it is flushed to the file system as a new SSTable. This is called a *minor compaction*. Since every minor compaction creates a new SSTable, the number of SSTables can get quite large and a read request may cause going and searching through all of these files. To prevent this, small SSTables are regularly converted into one large file, and unnecessary information such as deleted entries from the past are removed to gain space and decrease latency. **Figure 1.3** shows the basic tablet representation in BigTable.

Figure 1.3 – Tablet representation in BigTable: Each tablet server operates on several SSTable files as well as an in-memory table called memtable.
HBase is an open source implementation of BigTable licensed under Apache. It is based on the original BigTable paper, with minor variations in implementation. Like BigTable, its design goal is to store very large tables in hundreds of machines, running on top of a distributed file system and providing low latency read and write access. However, the terminology and the names of the building blocks used in the HBase architecture are different. Table 1.3 compares BigTable and HBase in terms of these naming conventions and implementation differences.

Table 1.3: Comparison of BigTable and HBase

<table>
<thead>
<tr>
<th>BigTable</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td>File System</td>
<td>Google File System</td>
</tr>
<tr>
<td>Storage File</td>
<td>SSTables</td>
</tr>
<tr>
<td>Basic Table Unit</td>
<td>Tablet</td>
</tr>
<tr>
<td>Locking Mechanism</td>
<td>Chubby</td>
</tr>
</tbody>
</table>

In our work, we use HBase excessively for storing phrase counts and likelihood estimates.

1.4.2 Distributed Caching and Memcached

Caching has been a common technique to exploit locality in computer applications. In the implementation of this project, we store the counts for all phrases and their likelihood estimates in a distributed database as described in the previous sub section. Our algorithms for chunking sentences require fetching those values from the database multiple times and using them for computation. Since each database query involves one or more round trips between the cluster nodes, caching the previously fetched values speeds up things and allows faster look-ups from the memory.

One common approach in caching is letting each node cache the records independently in its local dedicated memory. The problem with such a traditional approach is the limited memory on each machine and duplicating cached data on multiple nodes. Memcached is a distributed caching mechanism designed as an alternative to this limited approach. In a Memcached environment, each server dedicates some portion of its memory to a large, shared key space and the clients communicate with Memcached servers over a network to insert or fetch records. Unlike a distributed hash table, which may require travelling between multiple hops when locating a record, Memcached clients know which server they should communicate with in advance by the help of a hash function. Figure 1.4 briefly shows how a Memcached cluster operates.
Figure 1.4 – Operation of a Memcached cluster: Servers are located using a hash function and modulus operation.

Each server process initially allocates a particular amount of memory to cache the records and starts listening for requests. Client processes store a list of hostname and port number pairs for all of the servers. When a client wants to insert a record into the cache, it produces a server index to communicate with by performing the operation:

\[ s = \text{hash}(\text{key}) \mod n \]

where \( n \) is the total number of servers and \( \text{hash} \) is a universal hash function that maps a key to a number. Any client wanting to fetch the value associated with that key can perform the same calculation again and produce the index to locate the correct server. Memcached is designed as a dual layered hash table. The first layer is implemented in the client library to locate a server for a particular key. The second layer is in the server process, where a separate hash table is maintained in the local memory to store the key and value pairs.

Memcached uses a slab allocation technique for allocating memory. Large chunks of memory are previously allocated for different classes of key sizes varying from 64 bytes to 256 kilobytes, referred as slots. The server process maintains a list of the empty slots. When a request arrives for storing a key of a certain size, the smallest empty slot capable of storing that key is assigned. Similarly, when a record is removed from the cache, the corresponding slot is inserted back into the list of free slots. This mechanism prevents memory fragmentation and speeds up the look-up process.

1.5 Our Methods and Results

The contributions of this thesis fall into two areas: design, implementation, and evaluation of algorithms for chunking sentences into phrases, and methods for implementing such algorithms using popular distributed systems practices such as Map Reduce and Big Table.

In the first area, we present two new algorithms for discovering a most likely chunking into phrases of all sentences in a large corpus. The first algorithm is based on the assumption that every sentence is generated by randomly sampling from a dictionary which contains word unigrams, bigrams,..., n-grams for a large n value. It is modeled as an optimization problem and it requires iterating over the input data multiple times in order to maximize an objective function. The second method shares the same purpose, but it is established upon rather simple heuristics and requires only one pass through the input set aiming to reduce the computation time while achieving acceptable results. We compare these two methods in terms of accuracy, ease of implementation and complexity.

We deal with several implementation challenges as a result of working with a large data set and parameter rich models. Some challenges include generating and storing millions of phrases while maintaining fast read and write access times, frequently updating the records, caching common phrases in a distributed memory system to cut down database access rates, and iterating over this large collection of data multiple times.

In the second area, we describe methods to speed up the computation while making use of some popular and state of the art distributed systems practices such as Map Reduce, Big Table and Memcached. Our optimization techniques are adaptable to these paradigms and highly
exploit their special properties at many different stages. We identify the bottlenecks in the implementation and further focus on refining those parts, which include designing and evaluating multi-layered distributed storage mechanisms. Although we only apply these optimization techniques to the sentence chunking problem, we claim that they can be useful for other problems that share similar characteristics such as requiring common access to a large collection of \(<key,value>\) pairs and performing local search in a large state space with excessively many parameters.

Initial results with the heuristic rule based model produced acceptable chunkings, whereas we identified this model to be parameter-sensitive. We further found out that the statistical model is biased towards choosing longer phrases over short ones, and it needs a normalization factor to correct this behavior, which we will focus on as future work. Our experience suggests that the use of parallel computing tools dramatically reduce the execution time resulting in a good speed up. Experiments support the assertion that Map Reduce is an efficient programming model for text processing, shielding the programmer from having to worry about synchronization and communication issues while allowing better concentration on the main problem. We finally conclude that using a multi layered storage system that utilizes a Bloom Filter (20), local cache, and a distributed database performs well in terms high query throughput and serves well as a low-latency \(<key,value>\) look-up table while playing a significant role for scalability and performance.
Chapter 2

Background

We divide this chapter into two main sections as Theory and Systems. In the first section, we mainly focus on the previous work done in Linguistics, and review some problems similar to sentence chunking. The latter section concentrates on the use of Map Reduce as a tool in scientific problems including graph search, machine translation and topics in artificial intelligence.

2.1 Theory

2.1.1 Word Segmentation Problem

The process of discovering words from a long sequence of adjacent symbols without any indication of separation such as a space character is called the Word Segmentation Problem. Goldsmith (21) gives a more formal definition of this problem as follows:

Let $\Sigma$ be a fine grained alphabet containing the letters of a written language or the sounds of a spoken language, and let $L$ be a coarser set called a lexicon, which is a subset of $\Sigma^*$. We may assume that $L$ is our set of words. We further assume that every element of $\Sigma$ is also an element of $L$ to satisfy the condition that any element of $\Sigma^*$ is also an element of $L^*$. This means that each member of the alphabet is also a word and any string in $\Sigma^*$ will correspond to at least one member of $L^*$. We say at least, because frequently, a string in $\Sigma^*$ will correspond to more than one string in $L^*$. For example, the string atone corresponds to three natural sets of strings in $L^*$ as atone, at one and a tone, each of which is called a parse of atone.

Now let $C_1$ be a corpus where the words are separated by a space character. By a single pass, we can identify the lexicon $L$ and remove all spaces, yielding a new corpus $C_2$.

The Word Segmentation Problem asks this: Can we find a language independent algorithm $A_1(L, C_2)$ which can construct $C_1$ again?

It is still not known whether such an algorithm exists, and whether we can find it or not if it exists. A greedy solution approach to this problem starts scanning the text $S$ at position $i$ and finds the longest substring $s^*$ that occurs in the text again; then it skips to position $i + |s^*|$ and repeats the same procedure. The probabilistic approach uses a Markov model and chooses the highest probability string among all those in $L^*$ whose spell-out is $S$. 
2.1.1 Sentence Chunking has Similarities to Word Segmentation

In this subsection, we will take a closer look at the input and output of the Word Segmentation problem, and give a description of the Sentence Chunking problem in a similar fashion.

In the Word Segmentation problem, we can think of the alphabet $\Sigma$ as a set of letters of a written language. In this regard, the input and output can be depicted as such:

**Input**
1. A corpus $C_1$ consisting of letters from the alphabet $\Sigma$ without any indication of separation between the words.
2. A lexicon $L_1$, which in this case is a set of substrings generated from the alphabet $\Sigma$.

**Output**
1. A re-constructed sequence of letters with an indication of separation, typically a space character, at the word boundaries.
2. In the problem of Sentence Chunking, we can define a new alphabet $\Sigma_w$ as a set of words of a written language, rather than individual letters. The input and output of the problem then can be described as follows:

**Input**
1. A sequence of words from the alphabet $\Sigma_w$, with a clear indication of sentence boundaries by the appropriate use of punctuation marks. Briefly, this corresponds to a large corpus $C_2$ containing many sentences.
2. A lexicon $L_2$, which is a set of phrases generated from $\Sigma_w$. In other words, $L \subset\Sigma_w$.

**Output**
1. A valid chunking of each sentence into smaller phrases.
2. Remember that in the Word Segmentation example, there were three alternatives for segmenting "atone" as a tone, at one, and atone. Now consider the short sentence "this looks nice". We can chunk it in four different ways as this — looks nice, this looks — nice, this — looks — nice, and this looks nice. In this case we chunk a set of words forming a sentence to develop phrases, instead of chunking a sequence of letters to develop words. The major difference lies in the way we define the alphabets used to generate both streams.
2.1.2 Minimum Description Length Analysis

Another variation of the Word Segmentation Problem deals with a harder question. Given just
a long sequence of letters without any indication of separation between the words, can we
construct the lexicon? More formally, given $C_2$, can we find an algorithm $A_2(C_2)$ which
outputs $L$, where $L$ is the lexicon that we need for the first algorithm $A_1$?

Some interesting probabilistic approaches to this problem are based on Rissanen’s (22)
Minimum Description Length analysis which is the analysis made for finding the right
medium between two extremal encodings of the data. The first extremum is the case where $S$
is split into pieces at each letter boundary, assuming a letter corresponds to a word. The
second one is the opposite which suggests not splitting at all, implying that the whole string $S$
is just one very long word. The key point of MDL analysis is identifying the right encoding of
the text by taking two quantities into account: How much information do we use to encode a
text $S$, given a particular lexicon $L$; and how much information do we need to describe that
particular $L$?

Consider the first case, where we split $S$ at each letter boundary. Without loss of generality, if
we assume that $S$ is an English text, we need 26 characters to describe the lexicon. On the
other hand, given this short lexicon, we need to specify all of the subsequent letters in $S$ to
separate it from another text $S'$. In the second case the lexicon has one single element, so it is
easy to separate $S$ from $S'$, but that single item is so long that the description of the lexicon
requires too much information. In the desired medium where $L$ is constituted by the natural
words in $S$, the size of the lexicon is equal to the number of words, which is larger than 26,
but we also can separate $S$ from $S'$ by just spelling out subsequent words rather than letters,
which would require less information.

One question that quickly arises is how do we calculate the information required to encode the
data and describe a lexicon?

Given a particular lexicon, we can calculate the number of bits needed to encode a text $S$ by
$[-\log_2 P(S)]$, where the probability of $S$ is the product of its constituent words assuming a
probability distribution based on the lexicon in question. Calculating the number of bits
required to describe the lexicon is also simple. Considering an alphabet with $m$ letters, we
need $\log_2 m$ bits for each letter. If the lexicon has $N$ words, we can create a list of all words in
the lexicon using $\log_2 N + \sum_{i=1}^{N} |w_i| \log_2 m$ bits where $|w_i|$ is the number of letters in $w_i$.
Minimum Description Length analysis proposes that in the optimal case, the sum of these two
quantities forms a minimum.

2.1.3 Key Phrase Discovery

Key phrases are select terms extracted from text. Identifying key phrases in a document has
been an interesting research topic for its wide range of applications such as generating
metadata, fast skimming through text, indexing a collection of documents, and query
refinement in web search engines.
Turney (2) applied machine learning methods to the problem of key phrase discovery to extract a user defined number of key phrases from text using the C4.5 decision tree induction algorithm from Quinlan (23). He creates a list of all phrases of one, two or three words that appear in the text and uses a stemming method to generate stemmed phrases. Stemming in this context means dropping the suffixes from the end of a word to group words having common stems in the same phrase classes. For each stemmed phrase, he generates one feature vector with twelve features and creates a *soft-threshold* decision tree to classify each phrase as *positive* or *negative*. A *soft-threshold* decision tree contains a probability estimate for the class of each feature vector so at the end of the classification, the top $N$ vectors having the highest probability estimate are selected.

Turney further describes another system for automatic extraction of key phrases, GenEx (3), which is a combination of a parameterized set of heuristics, Extractor, and a genetic algorithm to tune its parameters, called Genitor.

Extractor is a deterministic algorithm involving multiple steps to extract key phrases. It relies on a number of parameters required to identify and score stem phrases, and do some post processing such as dropping duplicates, adding suffixes and adjusting capital letters.

Genitor is a steady state genetic algorithm (24) used to adapt the parameters for Extractor. Unlike a traditional genetic algorithm, a steady-state genetic algorithm changes one individual of the population at a time rather than creating an entirely new population, achieving a continuously change. Based on the experimental evaluations, Turney concludes that GenEx performs better than a custom designed C4.5 decision tree selection.

GenEx is showed to generalize well across collections. This means that it can be trained on a set of articles from domain $A$, and used to extract key phrases from a separate domain $B$. However, the training process is costly and requires a long time to correctly identify useful parameters.

Frank et. al (4) propose an alternative strategy to speed up the learning process when training and classification are performed on the same domain. They describe a naive Bayes learning algorithm to identify key phrases in a given text and apply it to unlabelled data. Initially, two attributes are defined and used for phrases up to length three: *TFxIDF score* and the *distance* into the document of the phrase’s first appearance.

*TFxIDF score* is used as a statistical measure to estimate the importance of a phrase $P$ to a given document $D$, calculated by the following:

$$TFxIDF(P,D) = Pr[\text{phrase in } D \text{ is } P] \times -\log Pr[P \text{ in a document}]$$

The first probability is calculated by counting the number of times $P$ occurs in the document $D$, and the second one is calculated by the number of documents containing $P$. The *distance* of a phrase $P$ is determined by counting the words that appear in the document $D$ before the first occurrence of $P$. Assuming these two attributes are independent, Frank et. al define $Pr[\text{key }| T,D]$ as the probability of $P$ being a key phrase given its *TFxIDF score* and *distance* as:

18
\[
\frac{Pr[T \mid key] \times Pr[D \mid key] \times Pr[key]}{Pr[T,D]}
\]

where \( Pr[T \mid key] \) is the probability that a key phrase has a TFxIDF score \( T \), \( Pr[D \mid key] \) is the probability that a key phrase has a distance \( D \), \( Pr[key] \) is the probability that a phrase is a key phrase and \( Pr[T,D] \) is the probability of a phrase having a TFxIDF score of \( T \) and distance \( D \). All these values are calculated by counting their occurrences in the training set.

### 2.1.4 Collocations Discovery

A relevant yet different problem is the identification of collocations in text using statistical methods. Although there is no single definition of a collocation, it generally refers to two or more words having a special property. Choueka (25) defines collocations as "A sequence of two or more consecutive words, that has characteristics of a syntactic and semantic unit". Smadja (26) defines a collocation as "A recurrent combination of words that occur more often than chance and that correspond to arbitrary word usages."

The easiest way to discover collocations would be to count all \( n \)-grams in text, and choose the ones with highest frequencies. However, this method is not very functional especially in the case of bigrams, because most of the time it discovers pairs of function words such as "of the", "in the", or "is a" as their counts are larger than that of the desired collocations like "stock prices" or "computer programming".

Justeson and Katz (27) improve this idea by applying a part of speech filter to the list of \( n \)-gram counts. For example, they accept all bigrams in the form of (noun, noun) or (adjective, noun) and reject the rest. This correctly identifies collocations such as "matrix multiplication" or "strong tea" whereas rejecting function words like "has been" or "at the".

It is not always necessary for the words of a collocation to appear subsequently in the text. An example would be "open ... window" as in "open a window", "open a new window", or "open a new browser window". Smadja (26) describes a mean and variance based method for discovering such collocations. The mean and variance are estimated by:

\[
\mu = \frac{1}{n} \sum_{i=1}^{n} d_i \quad \text{and} \quad \sigma^2 = \frac{\sum_{i=1}^{n} (d_i - \mu)^2}{n - 1}
\]

where \( n \) is the number of times the words occur together and \( di \) is the distance between co-occurring words for the \( i^{th} \) pair. This information is used to identify collocations by examining pairs with low standard deviations. The intuition behind this idea is that if two words form a collocation, they usually occur at about the same distance. If they are positioned closely a few times by chance, then most of the other times there will be a lot of words in between, and the variance \( \sigma \) will be higher stating those two words do not form a collocation.
Manning and Schütze (28) give example methods for discovering collocations by using *Hypothesis Testing* such as the *t-test* and *chi-square test*. In the *t-test*, they describe a null hypothesis $H_0$ indicating that two words $w_1$ and $w_2$ do not form a collocation given by:

$$P(w_1w_2) = P(w_1)P(w_2)$$

which implies that the occurrence of $w_1$ in the text is completely independent of $w_2$ and vice versa.

The *t-value* is estimated as:

$$t = \frac{\bar{x} - \mu}{\sqrt{\frac{s^2}{N}}}$$

where $\bar{x}$ is the sample mean, $s^2$ is the sample variance, $\mu$ is the mean of the distribution and $N$ is the sample size, which is the number of tokens in this case. Manning and Schütze (29) suggest that if the null hypothesis $H_0$ is true, then randomly generating bigrams and assigning 1 to the particular collocation $w_1w_2$ and 0 to the others is a *Bernoulli* trial with

$$p = P(w_1)P(w_2), \quad \mu = p, \quad \text{and} \quad \sigma^2 = p(1-p) \approx p$$

since for most bigrams $p$ is small. The probabilities and the sample variance are estimated by:

$$P(w_1) = \frac{c(w_1)}{N}, \quad P(w_2) = \frac{c(w_2)}{N} \quad \text{and} \quad \bar{x} = \frac{c(w_1w_2)}{N}$$

respectively, where $c$ indicates the number of occurrences of the corresponding token in the sample text. Once the *t-value* is calculated, the null hypothesis is either accepted or rejected and the bigram in question is considered as a collocation or not. In *chi-square test*, they describe a similar method to accept or reject a hypothesis without assuming normally distributed probabilities, which is shown to be true in general by Church and Mercer (1993).

Church and Hanks (30) define another metric, *point wise mutual information* $I$ as:

$$I(w_1,w_2) = \log_2 \frac{P(w_1,w_2)}{P(w_1)P(w_2)}$$

to measure the correlation between two words. Their assumption is that if two words form a collocation, then they occur more often together and are correlated. Notice that $I(w_1,w_2) \geq 0$ and it indicates the amount of information we have about $w_1$ occurring at position $i$ given the fact that we know $w_2$ occurs at position $i+1$ and vice versa.

The limitation with this approach is that in case of perfect dependency, $I(w_1,w_2) = \log_2 \frac{1}{P(w_1)}$, which becomes greater as the frequencies of the individual words get smaller. This behavior is not desired because it favors more information for less evidence. As a result, *point wise mutual
information is often considered as a good measure of independency, but it may be misleading when used to measure the dependency.

2.2 Systems

2.2.1 MapReduce in Scientific Applications

The design and development of parallel algorithms mostly rely on the underlying hardware architecture being used. Two common alternatives are expensive high-end servers offering peak single thread performance, and commodity clusters comprising a large number of mid-range desktop computers. The latter class of systems provides better performance per unit price, but is also more prone to failure.

In their description of the Google Search Architecture (31), Barroso et Al. showed significant evidence that the latter is more cost effective when supported with a software stack capable of taking care of individual node failures and data replication to increase the amount of parallelization and reliability. MapReduce (6) + HDFS (5) + HBase (7) is a distributed software stack designed to run on such clusters for large scale parallel application development.

MapReduce has been widely used as a parallelization scheme for a variety of applications including machine learning, statistical machine translation, graph search, and genetic algorithms. Recently, Ekanayake et Al. (32) (33) demonstrated that many scientific applications having SPMD characteristics can benefit from MapReduce to obtain scalability and speedup.

In this section, we discuss several examples of how MapReduce can be adapted to large scale scientific problems along with brief definitions of the problems and achieved speed ups. The background work summarized in this section is carefully chosen to cover some of the most common problems in MapReduce workflows including joining large data sets, performing a secondary sort on values and implementing custom partitioners to equally distribute the workload among the cluster nodes. Our work takes most of the ideas introduced in this section as a building block.

2.2.2 Machine Learning

Chu et. Al (17) described generalized methods to adapt existing machine learning algorithms to MapReduce without major modifications. Their idea builds upon extending the level of parallelism to those algorithms that fit Kearn’s Statistical Query Model (34), and exploit the parallel nature of the summation form. They observe that many machine learning algorithms require summation and multiplication over training vectors \( x_i \) and their labels \( y_i \), and suggest dividing these processes across multiple cores. They apply this parallelization idea to a wide range of machine learning algorithms with small implementation changes.

In the Locally Weighted Linear Regression (35) problem, the objective is to solve \( A\mu = b \), where \( A = \sum_i w_i(x_i^T x_i) \) and \( b = \sum_i w_i(x_i y_i) \). Two sets of mapper processes are used to calculate summations of subgroups \( \sum_{sg} w_i(x_i^T x_i) \) and \( \sum_{sg} w_i(x_i y_i) \). The intermediate sums are then aggregated inside two reducer processes to produce the final results.
The Backpropagation algorithm \((36)\) in Neural Networks is parallelized in a similar manner. Each mapper gets a subgroup of the training vectors and calculates a partial gradient by propagating the error through the network for each vector. Finally, a single reducer sums the partial gradients and performs a gradient descent to update the weights.

Naive Bayes \((37)\) requires estimating the parameters \(P(x_j = k \mid y = 1), P(x_j = k \mid y = 0)\) and \(P(y)\) by counting the corresponding events from the training data. To calculate the first two quantities, different sets of mappers sum over \(x_j = k\) for each \(y\) label and the intermediate pairs are grouped by \(k\). The reducers produce the final outputs \(P(x \mid y)\) by summing over the groups of records. The latter quantity \(P(y)\) is calculated by simply counting all occurrences of the events \(y = 0\) and \(y = 1\) with a MapReduce job again.

In k-means \((38)\), the input vectors are divided into subgroups and their distances to the centroid vectors are calculated inside the mappers. Each input vector is then assigned to the group of the corresponding centroid where its distance is minimum. The calculation of the new centroid vectors requires an extra MapReduce job. In the map phase, the input vectors are divided into subgroups and partial sums for each subgroup are calculated in parallel. At the reduce step, these partial sums are added and the new centroid vectors are found by taking their average.

In Principal Component Analysis \((39)\), the objective is to compute the eigenvectors of the covariance matrix \(A - B\) where \(A = \sum_i^n x_i x_i^T\) and \(B = \mu \mu^T\). The mean vector \(\mu\) can further be expressed as \(\mu = \frac{1}{m} \sum_{i=1}^m x_i\) so that both \(A\) and \(B\) can be computed in parallel, again by dividing the input vectors \(x_i\) in subgroups to calculate partial sums in the mappers, and aggregating the intermediate data in the reducers.

Chu et al. compare the performance of these algorithms with their serial counterparts and obtain linear speed up rates. They show 1.9 times speed up on a dual core machine, and up to 54 times speed up on 64 cores. They conclude that implementing existing machine learning algorithms with MapReduce by utilizing the distributed structure of the summation form allows easy parallelization and results in linear speed-ups.

### 2.2.3 Statistical Machine Translation

Dyer and Cordova \((19)\) present applications of MapReduce to Statistical Machine Translation Models aiming to scale them for large training sets. Their motivation is based on the observation that the translation quality improves by increasing the size of the input, which brings the additional cost of long running times and extra storage requirements. They present MapReduce implementations of a phrase based translation model and word alignment models by pair wise lexical translation, demonstrating significant speed up that reduce the running times from several days to hours. The main contribution of their work is the description of three general methods to compute the maximum likelihood estimates for finite, discrete probability distributions.

The maximum likelihood criterion is often used to estimate conditional probability distributions over discrete events as follows:
\[ P(B|A) = \frac{c(A, B)}{c(A)} = \frac{c(A, B)}{\sum_{B'} c(A, B')} \]

In the first method, three Map Reduce jobs are run sequentially. The first computes \( c(A, B) \) for all pairs, the second computes the marginal counts \( c(A) \) for all \( A \)'s, and the third groups all values for a given \( A \) to compute the conditional probabilities.

The input and output pairs of these jobs are summarized in Table 2.1.

<table>
<thead>
<tr>
<th>Map Reduce</th>
<th>(A,B) =&gt; ((A,B),1) [ key: (A,B) ] value: c(A,B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Reduce</td>
<td>((A,B),c(A,B)) =&gt; (A,c(A,B)) [ key: (A,nil) ] value: c(A,nil)</td>
</tr>
<tr>
<td>Map Reduce</td>
<td>((A,B),c(A,B)) =&gt; ((A,B),c(A,B)) [ key: (A,B) ] value: ( \frac{c(A,B)}{c(A)} )</td>
</tr>
</tbody>
</table>

One important detail of this method is that the third reducer requires a secondary sort on values, which is a common problem in MapReduce applications. Notice that the third mapper emits (key,value) pairs of the form (A.x), where x can either be an event representing the count for a particular pair such as (A.B) or the marginal count for A, (A.nil). The third reducer needs to obtain the marginal count first, store it in the memory and process the remaining pairs. Thus, the framework needs to ensure that ((A,nil),c(A,nil)) is the first pair the reducer receives. This can be achieved by sorting the values for a particular event \( A \) and passing them to a single reducer. For example, if there are three (key,value) pairs as ((A.D),c(A,D)), ((A,nil),c(A,nil)) and ((A.B),c(A,B)), we want to guarantee that the reducer receives them in the following order: ((A,nil),c(A,nil)), ((A,B),c(A,B)) and ((A.D),c(A,D)).

An advantage of the MapReduce framework is that it already sorts the keys before passing them to the reducer. However, each separate key goes to a different group by default. This means the pairs ((A,nil),c(A,nil)), ((A,B),c(A,B)) and ((A.D),c(A,D)) would be assigned to three different groups. This default behavior is changed by writing a custom comparator to be used during the group creation phase. Instead of comparing the whole key (A.x), the comparator only takes into account the first part before the dot symbol to send all the keys starting with a particular \( A \) to the same reducer. Since those keys starting with a particular \( A \) are already sorted, the reducer receives the values in sorted order.

The second method is similar but only requires one MapReduce job to compute the likelihoods. The mapper process emits two (key,value) pairs instead of one for a particular event (A,B): ((A,B),1) and ((A,nil),1). The second pair is used to calculate the marginal count c(A). The trick for sorting the values in the first method is again used so the reducer gets the pairs associated with the marginal count first and sums them up.

The advantage of the second method over the first is that it only requires one MapReduce job. However, it generates and processes more intermediate data.
Table 2.2 summarizes the input and output pairs for this job.

**Table 2.2: Input – output pairs for maximum likelihood estimate calculation – alternative 2**

<table>
<thead>
<tr>
<th>Map</th>
<th>(A, B) → ((A, B), 1), ((A, nil), 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce</td>
<td>key: (A, B) value: ( \frac{c(A, B)}{c(A)} )</td>
</tr>
</tbody>
</table>

In the third method, the counts of all B events associated with a particular A = a are sent to the same reducer. This method has an extra requirement that the intermediate pairs for \( P(B|A) \) fit inside the memory. All of the pairs \((A, (B_i, 1))\) are stored in a random access data structure in the memory and the marginal count for A is computed by a single pass. The counts for individual pairs are divided by \( c(A) \) to compute the likelihood estimates in a second pass. The input and output pairs are specified in Table 2.3.

**Table 2.3: Input – output pairs for maximum likelihood estimate calculation – alternative 3**

<table>
<thead>
<tr>
<th>Map</th>
<th>(A, Bi) → (A, (Bi, 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce</td>
<td>key: (A) value: ( \frac{c(A, B_1)}{c(A)}, \frac{c(A, B_2)}{c(A)} )</td>
</tr>
</tbody>
</table>

The third method also enables the use of combiners[ref-to-thesis] to reduce the amount of intermediate data and the communication costs. Dyer and Cordova compare the performance of these three methods and demonstrate that the third one performs up to twice as fast, concluding to use it for the statistical translation models.

2.2.4 Graph Search

Kang et Al. (18) estimate the effective diameter of massive graphs while describing how large data sets can be joined by Map Reduce. The effective diameter is defined as the minimum number of hops in which 90% of all connected pairs of nodes can reach each other. They define \( N(h) \) as the number of node pairs \((a, b)\) such that \( b \) is reachable from \( a \) within \( h \) hops; and \( b(h, i) \) as the bitmask of node \( i \) which contains the nodes that are reachable from \( i \) within \( h \) hops. An iterative MapReduce workflow is described to calculate \( N(h) \) for \( h = (1, 2, \ldots) \) until some stopping condition is met.

The input to the program is an edge file \( E \) containing (sourceId, destinationId) pairs and a bitmask file \( B \) that holds the bitmasks for each vertex in the graph. \( B \) is created at the first iteration for \( h = 1 \) based on the pair wise adjacency relationships specified in \( E \), and updated at every iteration for increasing values of \( h \). The update routine is built on the following observation:

\[
(b(h + 1), i) = b(h, i) \text{ BITWISEOR } \{b(h, k) \mid E.sourceId = i \text{ AND } E.destinationId = k\}
\]

They describe a naive implementation, in which each mapper receives and outputs (sourceId, destinationId) pairs, and each reducer collects all of the edges that originate from a particular source vertex \( v \). Then the reducers download the bitmask file \( B^h \) from the distributed file system and apply the update routine to compute the new set of bitmasks \( B^{h+1} \) for iteration \( h + 1 \). Upon completion, another MapReduce job is run to compute the neighborhood function \( N(h + 1) \) and a decision is made to whether or not to proceed to the next iteration.
The naive implementation has performance limitations due to redundant copying of the bitmask file $B$. Consider the simple scenario where there are only two edges in the graph as $(v_1, v_3)$ and $(v_2, v_4)$. In this case, the reducer processing $v_1$ does not need to download the bitmask data for $v_2$ and $v_4$; similarly, the reducer processing $v_2$ does not need the bitmask data for $v_1$ and $v_3$. Ideally, the reducers should only get the bitmask data they need: $v_1$ needs $b_3$ and $v_2$ needs $b_4$, where $b_i$ is the bitmask data associated with the node $i$.

![Diagram showing redundant data in the operation of HADI]

Kang et al. suggest performing a join operation to achieve this and avoid redundant copying. Essentially, they run a two stage MapReduce job to join the edge file $E$ with the bitmask file $B$ interpreting the `sourceId` as the key.

In the first map, there are two kinds of input data, $(v_i, v_j)$ for the edges and $(v_i, b_i)$ for the bitmasks. The mappers invert the edge pairs $(v_i, v_j)$ as $(v_j, v_i)$ and pass them to the reducers. The bitmask pairs are directly passed to the reducers without being modified. At the reduce step, the bitmasks $(v_j, b_j)$ are grouped together with the inverted edges $(v_j, v_i)$. For each such edge-bitmask pair, the reducers output $(v_i, b_j)$, which is the pair of interest since processing $v_i$ requires the bitmask $b_j$, due to the edge $(v_i, v_j)$ in the original graph. The second MapReduce job groups together all bitmasks required for processing a particular vertex $v_k$, and applies the `BITWISEOR` operation to update the its bitmask. Once the new bitmasks are updated for all vertices in the graph, a third MapReduce job is run to compute the neighborhood function $N(h)$.

Kang et al. show that their implementation is close to the ideal scale up until they hit some bottlenecks such as the network or JVM load times. They further use their system to estimate the diameter of the largest web graph and disprove the conjecture of Albert et al.[5] about the web’s diameter.

### 2.2.5 Genetic Algorithms

Verma et Al. (40) adapt Genetic Algorithms into MapReduce and demonstrate its scalability to large problems containing up to 105 variables. They suggest performing each iteration as a separate MapReduce job and pulling the latest results from a controlling client at the end of each iteration. The client is responsible for deciding when to stop the iterations, based on the...
fitness of the best individual in the latest population. The mapper processes evaluate the fitness value of the individuals in the current population, and pairs in the form of (individual, fitness-score) are passed to the reducers for further recombination.

Unlike a traditional Map Reduce workflow in which, pairs sharing the same key are sent to the same reducer, a custom partitioner is implemented to assign intermediate pairs to random reducers. The motivation for a random partitioner is to ensure that the genetic algorithm converges while utilizing all the processing power in the cluster. Since the default partitioner groups the pairs based on hash(key), it introduces an artificial spatial constraint and isolates the individuals within each partition from the rest of the population. This may reduce the speed of convergence, or prevent the genetic algorithm from converging to the global optimum.

Another side effect of the default partitioner stems from the associative nature of the Genetic Algorithms. As the population evolves and the genetic algorithm progresses, the same individual who is close to the optimum solution begins to dominate the population, creating replicas of itself. If the default partitioner is used, as the algorithm converges almost all of the individuals are sent to a single reducer, resulting in heterogeneous workloads among reducers, and thus the cluster becomes non-utilized. The difference between the two partitioners is given in Table 2.4.

<table>
<thead>
<tr>
<th>Default</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = hash(key) % r</td>
<td>p = rand % r</td>
</tr>
</tbody>
</table>

The use of the random partitioner gives rise to complete shuffling of the individuals across the population, increasing genetic diversity and reducing the number of iterations required for convergence.
Chapter 3

Design and Implementation

3.1 Chunking Models

In this section, we introduce two algorithms to chunk sentences into their constituent terms and discuss the applicability of these algorithms to large data sets using parallel programming methods. The first algorithm is based on a statistical model, and requires multiple iterations to maximize an objective function. The second algorithm is built on simple heuristic rules that take into account how many times a phrase occurs in the corpus as well as how many words it contains, and assigns weighting factors to each phrase based on these two parameters.

3.1.1 Statistical Model for Sentence Chunking

This model is based on the assumption that every sentence is generated by randomly sampling from a dictionary which contains word unigrams, bigrams, ..., n-grams for some value $n$. Since each sentence can be partitioned (chunked) into its phrases in a finite number of ways, our model assigns a likelihood value to each phrase, and chooses the chunking with the maximum likelihood. We make sure that the sum of all likelihood values assigned to a phrase in the corpus is equal to 1, and each individual likelihood value is between 0 and 1.

$$\sum_{j=1}^{\infty} f_j = 1, \ 0 \leq f_i \leq 1$$

The likelihood value associated with a single chunking is computed by simply multiplying the likelihood of all phrases that constitute it.

$$P(C_i|\{f_k\}_{k=1,..,m}) = \prod_{j \in C_i} f_j$$

The probability of a sentence is then given by the following:

$$P(sentence = 1,2,3,4|f_k) = f_1 f_2 f_3 f_4 + f_1 f_2 f_3 f_4 + f_1 f_2 f_3 f_4 + f_1 f_2 f_3 f_4 + f_1 f_2 f_3 f_4 + f_1 f_2 f_3 f_4 + f_1 f_2 f_3 f_4 + f_1 f_2 f_3 f_4$$

The probability of each solution, however, is contingent on the arrangement of words in phrases $f$ and sentences $S$ throughout the corpus as follows:

$$P(\text{Corpus} = \{sentence_i\}_{i=1,..,M}|\{f_k\}_{k=1,..,m}) = \prod_{i \in \text{Corpus}} P(sentence_i|\{f_k\}_{k=1,..,m})$$
We then take the maximum likelihood estimates for parameters $\Theta$ that correspond to the expected frequency of each phrase $\{f_k\}_{k=1,\ldots,\kappa}$. (Note that these frequency estimates are distinct from and should not be confused with the observed n-gram frequencies):

$$\theta_{ML} = \arg\max_\theta [P(\text{corpus}|\theta)]$$

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

Maximize

$$f(\text{A}) \cdot f(\text{BC}) + f(\text{A}) \cdot f(\text{B}) \cdot f(\text{C}) + f(\text{AB}) \cdot f(\text{C}) + f(\text{ABC}) \cdot f(\text{A}) \cdot f(\text{B})$$

$$+ f(\text{AB}) \cdot f(\text{A}) \cdot f(\text{BD}) + f(\text{A}) \cdot f(\text{B}) \cdot f(\text{D}) + f(\text{AB}) \cdot f(\text{D})$$

Subject to

$$\sum_{j=1}^{\rho} f_j = 1, \text{ and } 0 \leq f_i \leq 1$$

Fast computation of sentence probabilities: Computing $P(\text{sentence})$ involves summing the likelihoods of all possible chunkings, which grows exponentially with the number of words in the sentence. If there are $n$ words in a sentence, the number of chunking alternatives becomes $2^n - 1$.

However, we can easily compute $P(\text{sentence})$ with a canonical Viterbi algorithm. As an example, consider a four-word sentence again. We can compute the sentence probability as follows.

$$f_{\text{Begin}} \leftarrow 1$$
$$f_{\text{End}} \leftarrow 1$$

$$f_i \leftarrow f_i \cdot \sum_j \{P_j | j \text{ is connected to } i \text{ by arc}(j, i)\}$$

$$P_{\text{Begin}} \leftarrow 1$$
$$P_1 \leftarrow f_1$$
$$P_{12} \leftarrow f_{12}$$
$$P_{123} \leftarrow f_{123}$$
$$P_{1234} \leftarrow f_{1234}$$
$$P_{234} \leftarrow f_{234}P_1 = f_1f_{234}$$
$$P_{23} \leftarrow f_{23}P_1 = f_1f_{23}$$
$$P_2 \leftarrow f_2P_1 = f_1f_2$$
$$P_3 \leftarrow f_3(P_1 + P_{12}) = f_3(f_1f_2 + f_{12})$$
$$P_{34} \leftarrow f_{34}(P_2 + P_{12})f_{34}(f_1f_2 + f_{12}) = f_{34}(f_1f_2 + f_{12})$$
The algorithm is a relative of the backward-forward algorithm used for hidden Markov models and stochastic parsers based on context-free grammars.

**Improving the model** When working with the initial model, we discovered that it lacks a normalization factor to produce the expected results. When computing the probability of a chunking $C_i$, we simply multiply the likelihoods $f_j$ of all phrases $j \in C_i$. This has a negative effect on the optimization problem formulation because multiplying two probabilities $p_i$ and $p_j$ always results in a smaller number since $0 \leq p_i, p_j \leq 1$. As a result, the initial model always tends to choose larger phrases over smaller ones, which gives rise to an unwanted behavior by always selecting the whole sentence as one big chunk, instead of dividing it into smaller partitions.

Consider a simple sentence with three words. The initial model tends to choose $f_{12}$ over $f_1f_2$ and similarly $f_{123}$ over $f_{12}f_3$. When the algorithm terminates, we simply end up with sentences not being partitioned into smaller chunks. In order to tackle this problem, we use a normalization parameter $l_i$ which is simply a function of the number of words $n_i$ in phrase $f_i$.

$$l_i = \frac{n(i)^{-\gamma}}{\sum_{j=1}^{\text{max}} j^{-\gamma}}, \gamma > 1$$

where $\text{max}$ is the length of the phrase with the maximum number of words in the corpus, and $\gamma$ is a constant. The probability of a chunking is calculated as follows:

$$P(C_i|f_k)_{k=1,\ldots,n} = \prod_{j \in C_i} f_j l_j$$

Observe that the idea is quite simple. We multiply the likelihoods of longer phrases with smaller constants, and shorter phrases with larger constants. As an example, consider a phrase with two words. We now expect the model to make a decision between $f_{1,2}(l_2)$ and $f_1f_2(l_1)^2$, given $(l_1)^2 > (l_2)$ for $\gamma > 1$ so that the model is not biased against larger chunks.

**Reducing the problem size:** We do some optimizations to reduce the number of parameters while approximating in some way. We assume phrase frequencies belong to $C$ distinct classes and have one parameter for each class. Then, each phrase belongs to a specific frequency class based on its count (frequency) over the whole corpus.

For example, all of the phrases that occur only once belong to $\text{class}1$, and similarly all the phrases that occur twice in the corpus belong to $\text{class}2$, and so on.
If $n_i$ and $f_i$ show the number of phrases, and the likelihood value associated with $\text{class}_i$ respectively, it follows that:

$$\sum_{i=1}^{C} n_i f_i = 1$$

Observe that while grouping phrases inside frequency classes reduces the number of parameters in the problem domain, we still have to keep track of how many times each phrase occurs in the corpus. As an example, suppose we are interested in looking-up the likelihood value associated with the phrase “medicine”. We need to perform two look up operations to achieve this. First, we retrieve the count $k$ “medicine” by querying a storage system. Second, we issue another query to retrieve the likelihood value $f_k$ associated with class $k$.

**Hill Climbing:** To maximize the objective function, we use a hill climbing algorithm by randomly altering some percentage of the likelihoods and computing $P(\text{Corpus})$ at each step. If a stopping criterion is met, the algorithm terminates. Various stopping criteria can be implemented to check if the algorithm should perform one more iteration or not. In our experiments, we used the average number of words as the stopping criterion. The algorithm terminates if the average number of words per phrase is larger than some threshold $t$.

The pseudo code of the algorithm is given in the following listing.

```
Chunk Sentences
Initialize the currentSolution randomly
max = − infinity
bestSolution = currentSolution
loop
    $P_{\text{corpus}} = \prod_{i=1}^{N} P(\text{sentence}_i)$
    if $P_{\text{corpus}} > max$ then
        max = $P_{\text{corpus}}$
        bestSolution = currentSolution
    else
        if stopping criteria is met then
            break
        else
            currentSolution = alter(bestSolution)
        endif
    endif
end loop
```
3.1.2 Heuristic Rule Based Model for Sentence Chunking

The heuristic model is built on two simple observations regarding the chunking problem.

1. Phrases with higher frequencies are more likely to become individual chunks of a sentence.
2. The maximum number of words in a phrase is bounded by some artificial threshold $t$. In other words, it intuitively makes sense to divide a phrase having $n$ words into two smaller phrases containing $n_1$ and $n_2$ words respectively, where $n=n_1+n_2$ and $n \geq t$.

This model does not require multiple iterations over the corpus. Instead, we associate each phrase $f_i$ with a weight factor $w(f_i)$ based on its frequency and the number of words it has. The score of a chunking $C_i$ is computed by simply summing up the weights of the phrases that constitute it.

$$w(C_i) = \sum_{f_j \in C_i} w(f_j)$$

Since each directed walk in the phrase graph corresponds to a different chunking, the optimum chunking for a sentence can be computed by traversing the phrase graph and looking for the maximum weight path (critical path) in polynomial time. For example, consider a sentence with three words. The critical path is given by:

$$\text{max} \left( w(f_1) + w(f_2) + w(f_3), w(f_{12}), w(f_{123}) \right)$$

The weight of a phrase $f_i$ containing $n_i$ words and having a frequency of $k_i$ is computed by:

$$w(f_i) = \alpha(k_i) \beta(n_i)$$

where, $\alpha$ and $\beta$ are two normalization filters applied to the weighting factors.

The $\alpha$ filter is just a $\lambda$ shaped function, which linearly increases the weight of a phrase up to a frequency threshold $t$. If the frequency of the phrase is more than $t$, the filter starts decreasing the weight. Finally, if the phrase is quite common in the corpus (such as "the", "or", "a"), the $\alpha$ filter returns a 1.

$$\alpha(x, t) = \begin{cases} x, & \text{if } x < t \\ -x + 2t, & \text{if } t < x < 2t \\ 1, & \text{otherwise} \end{cases}$$

The $\beta$ filter is quite similar to $\alpha$ in terms of functionality. It increases the weight of a phrase up to a length threshold $t$. (In this context, length refers to the number of words in a phrase). If the length is more than $t$ the filter starts decreasing the weight. Again, if the length of a phrase is too large, the $\beta$ filter returns a 1. Contrary to the $\alpha$ filter, $\beta$ increases and decreases the weight of a phrase exponentially and its shape is similar to a Gaussian.

$$\beta(x, t) = 2^{k(\alpha(x, t) - 1)}$$
The shapes of these two functions are seen in the following figures:

**Figure 3.1 - Shape of the $\alpha$ filter for $t = 1500$:** Observe that we linearly increase the weight of a phrase up to the threshold value, and then begin to decrease it. The insight is to select common phrases over the rare ones while eliminating those having top frequencies, which fall into the tail area and are generally of length one.

**Figure 3.2 – Shape of the $\beta$ filter for $t = 3$:** We assume that in general a phrase length falling in the [1-5] range is a good choice for the length of a chunk. If the length of a chunk is greater than five, it makes more sense to divide it into two smaller chunks. In addition, examine that the $\beta$ filter has a steeper shape because our model is stricter on the length of a phrase then its frequency in the corpus.
3.2 Implementation

The Implementation stage can be divided into the following subroutines each of which corresponds to one or more Map Reduce applications:

- Counting phrases
- Creating a histogram of phrase counts
- Filtering phrases with \( count > 1 \)
- Uploading phrase counts to a distributed storage system
- Running and monitoring the sentence chunking application

In this section, we will explain each of these subroutines and discuss their implementations using Map Reduce.

3.2.1 Counting Phrases

Phrase count is a simple variation of the widely known word count[45] application which perfectly fits the Map Reduce programming model. The input to the program is a large corpus in the form of text files, where each line is a sentence ending with a punctuation mark.

Each mapper process receives a sentence \( S_k \) as input, and generates all possible phrases \( p_j \) \( p_j \in S_k \), emitting them in the form \( \langle p_j, 1 \rangle \).

Combiners are used to group \( \langle key, value \rangle \) pairs \( \langle p_j, \{ 1, \ldots, 1 \} \rangle \) and generate an intermediate count \( \langle p_j, c_j \rangle \) before shipping them to reducers.

Finally, reducers receive groups of records \( \langle p_j, \{ c_{j1}, \ldots, c_{jn} \} \rangle \) and produce the final count \( \langle p_j, c_f \rangle \) such that \( c_f = \prod_{i=1}^{n} c_{ji} \).

We use the same function for the combine and reduce steps since they are functionally equivalent tasks. Table 3.1 shows the input and output of these jobs.

<table>
<thead>
<tr>
<th>Table 3.1: Input-output pairs for Phrase Count application</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Map</strong></td>
</tr>
<tr>
<td><strong>Combine</strong></td>
</tr>
<tr>
<td><strong>Reduce</strong></td>
</tr>
</tbody>
</table>
The pseudo-code for the map and reduce functions are as follows:

### Phrase Count

**Map** (key `null`, value `S_i`):

```
foreach phrase p_j in S_i
    emit(p_j, 1)
```

**Reduce** (key `p_i`, values [{ `c_1`, ..., `c_n` }]):

```
sum = 0
foreach c_i in values
    sum += c_i
emit(p_i, sum)
```

The time complexity of this procedure depends on the average number of words per sentence and the number of sentences in the corpus. It grows linearly with the size of the input since it only requires a single pass over the input set. An interesting outcome of this application is the distribution of the phrases and their counts. If we observe the output of the reduce step, we see that most phrases have a count of 1. This is actually not surprising because the map process generates every single phrase contained in a sentence, and most of those phrases, especially the longer ones, are local to the sentence that they belong to. In order to visualize this, we generate a histogram next, showing the number of phrases with a specific count.

#### 3.2.2 Creating a histogram of phrase counts

The input to this task is simply the output of the phrase count application. Each mapper receives <key, value> pairs of the form < `p_j`, `c_j` > and emits a single record < `c_j`, 1 >.

Again, the combiners just group <key, value> pairs < `c_j`, {1,...,1} > and generate an intermediate count < `c_j`, `f_j` > before shipping them to reducers.

Reducers then receive groups of records < `c_j`, {`f_{j1}`, ..., `f_{jn}`} > and produce the final count < `c_j`, `f` > such that `f = \sum_{i=1}^{n} f_{ji}`, where `f` shows the number of phrases that occur `c_j` times in the corpus. We use only one reducer to have a single output file containing the whole distribution. Notice that it doesn’t induce a significant performance loss since most of the reductions occur locally during the combine step. **Table 3.2** summarizes the input and output of these computations.
Table 3.2: Input-output pairs for computing a histogram of phrase counts.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Map</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>((p_j, c_j), \rightarrow (c_j, 1))</td>
</tr>
<tr>
<td><strong>Combine</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>((c_j, {1, \ldots, 1}), \rightarrow (c_j, f_j), c_j \geq 1)</td>
</tr>
<tr>
<td><strong>Reduce</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>((c_j, {f_{j1}, \ldots, f_{jn}}), \rightarrow (c_j, \sum_{i=1}^{n} f_{ji}))</td>
</tr>
</tbody>
</table>

The pseudo-code for the map and reduce functions is as follows:

**Histogram**

**Map** (key \(p_j\), value \(c_j\)):
emit\((c_j, 1)\)

**Reduce** (key \(c_i\), values[ ] \(\{f_1, \ldots, f_n\}\)):
sum = 0
foreach \(f_i\) in values
sum+ = \(f_i\)
emit\((c_i, \text{sum})\)

Visualizing the graph of the output pairs shows that the data seem to follow the Zipfian distribution. This means if we rank the phrases based on their frequencies, the rank of a phrase becomes inversely proportional to its frequency. Moreover, there are only a few phrases that occur quite frequently, whereas most of the phrases occur only once in the corpus. In our 240 MB of sample corpus containing scientific papers as plain text, we found that approximately 98% of the phrases occur only once in the corpus.

![Figure 3.3 – Distribution of Phrase Frequencies](image-url)

**Figure 3.3 – Distribution of Phrase Frequencies:** There are only a small number of phrases that are very common in the corpus. 98% of them occur only once.
3.2.3 Introducing Problems of Scale

From a systems perspective, we exploit this nature of the data to reduce the amount of information stored in our distributed storage system. Both of our chunking models require knowing the frequency of each phrase during computation. This information is stored either in a distributed memory or a distributed database mechanism, and we simply query this system at least once for each unique phrase in the corpus.

Storing the frequency of each phrase in a distributed storage system is a costly operation in terms of space. The output of the phrase count application is large because it populates the input data when generating the phrases. For example, a relatively small corpus of size 200 MB generates 50 GB of <phrase, count> pairs as output. Querying such a large database becomes another challenge, since every read request triggers a series of look-up operations such as scanning table indices and rows to fetch the desired data block (section 1.4.1).

We address this problem by simply storing only the phrases having count > 1. When querying the storage system to fetch the frequency of a particular phrase \( p_i \), the look-up operation returns either a value \( c_i > 1 \) or a null reference. A null reference indicates that we never stored \( p_i \)'s frequency implying \( c_i = 1 \). The pseudo-code for this operation is as follows:

```
Phrase Frequency Look-up

Input: (phrase \( p_i \)):
value = table.get(\( p_i \))
if (value = null)
    return 1
else
    return value
```

This optimization has two immediate consequences. It dramatically reduces the amount of data needed to store in the distributed storage system, and read requests are issued faster since queries need to scan only a small subset of the data.

3.2.4 Filtering phrases with count > 1

We filter phrases with count > 1 by a simple map operation, without a reduce step. Each mapper receives input pairs \( < p_i, c_i > \) and emits if \( c_i > 1 \). A reduce step is not necessary because we don't need to group the records by their keys to apply the filter. The output from the mappers is directly written to files in HDFS.

Figure 3.4 shows the relationship between the size of the corpus, and the size of the filtered phrase frequency data, which only contains those phrases having count > 1. The filtered data grows linearly with the corpus size, unlike the output of the phrase count application which produces large amounts of \( < p_i, c_i > \) pairs, where in most cases \( c_i = 1 \).
3.2.5 Initial implementation with HBase

We describe the implementation of both models and highlight some of the technical differences in this section.

3.2.5.1 Statistical Model

Our objective in this model is to compute P(Corpus), the probability associated with the input data set. Each mapper receives sentences $S_i$ as input, and computes the sentence probabilities $P(S_i)$ independently, emitting an intermediate output pair $<1, P(S_i)>$.

A notable remark is the use of a single key (1), as the output of the mappers since we are interested in computing the corpus probability, which is the product of all sentence probabilities as described in section 3.1.1.

In addition, mappers constantly query the distributed storage system to fetch the frequency $c_j$ of each phrase $p_j$. Once $c_j$ is obtained, the phrase is associated with a frequency class $k$, and a corresponding likelihood value $f_k$ is used for computing $P(S_i)$.

We don't store the <frequency class, likelihood> relation in a distributed storage system because this information can easily fit in a single binary file, and its size is in the order of a few hundred kilobytes. This file is stored under HDFS, and each map process reads its contents into their local memories before starting the actual computation. At each iteration, a controlling process updates this file and a new likelihood distribution is generated.
Combiners simply bring together all the records as one group, and produce a local product of sentence probabilities. The output of a combiner \( c \) receiving \( n \) sentences \( \{S_1, \ldots, S_n\} \) is just the pair \( P(S_c) = \langle 1, \prod_{i=1}^{n} P(S_i) \rangle \).

We use only one reducer to produce a single output file with just one record \( P(\text{Corpus}) \) as the output. The reducer is functionally equivalent to the combiners, and it computes the final product by multiplying the intermediate products \( P(S_{c_i}) \) received from the combiners. The output of the reducer is the single record \( P(\text{Corpus}) = \prod_{i=1}^{n} P(S_{c_i}) \).

Table 3.3 shows the input and output of the map, combine, and reduce stages.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>((S_i), \rightarrow (1, P(S_i)))</td>
<td></td>
</tr>
<tr>
<td>Combine</td>
<td>((1, {P(S_1), \ldots, P(S_n)}), \rightarrow (1, \prod_{i=1}^{n} P(S_i)))</td>
<td></td>
</tr>
<tr>
<td>Reduce</td>
<td>((1, {P(S_{c_1}), \ldots, P(S_{c_n})}), \rightarrow (1, \prod_{i=1}^{n} P(S_{c_i})))</td>
<td></td>
</tr>
</tbody>
</table>

### 3.2.5.2 Heuristic Model

This model does not require any aggregation so we omit the use of combiners and reducers. The algorithm can be implemented as a simple embarrassingly parallel program, where each mapper receives sentences \( S_i \) as input, and computes the maximum weighted path for each sentence, finally emitting a chunked sentence as output (section 3.1.1). As in the statistical model, each mapper constantly queries the distributed storage system to fetch the frequency \( c_j \) of each phrase \( p_j \).

### 3.2.6 Uploading Phrase Counts to a Distributed Storage System

We describe several alternative distributed storage mechanisms to be used as a \(<\text{key}, \text{value}>\) look-up table by the sentence chunking applications.

#### 3.2.6.1 HBase

In our initial implementation, we used HBase (7) as our distributed storage system and utilized both sentence chunking models on top of a phrase frequency table. Profiling the initial implementation pointed out that a considerable amount of time is spent during the phrase count look-ups. This is because each phrase count request goes over the network, and the minimum time to fetch the count of a phrase is equal to the network round trip time. Also, HBase maintains a sorted index of row keys in a table and each read request requires performing a binary search over a subset of this indexed structure. These operations introduce some additional delay depending on the number of concurrent requests and the size of the phrase count table.

In order to address this bottleneck, we designed and evaluated five alternative storage mechanisms to perform the phrase frequency look-up operations for use in the chunking algorithms.
3.2.6.2 Memcached

We used Memcached (9) to reduce the database search overheads introduced in HBase. Every record in Memcached is a <key, value> pair, stored in the memory, contrary to HBase which is a persistent storage system that frequently utilizes the hard drive. This implementation assumes that the < phrase, count > relation is small enough to fit into a distributed cache, which is a realistic assumption given the size of the filtered phrase count data set. Moreover, since Memcached simply distributes the data over multiple nodes, we can throw in more nodes whenever we need to scale the system and work on a larger problem.

Memcached is typically and widely used for caching persistent data in web applications. Whenever a Memcached look-up fails, the client application queries a backend database server to fetch the data. In our implementation however, Memcached is basically treated like an in memory database. We insert all phrases with a count > 1, and in case of a cache miss, our client applications (mappers) assume that the frequency of the phrase is 1. We carefully monitor all of the Memcached nodes during computation and ensure that none of them fails to ensure that our computations are accurate. When starting the Memcached servers, we disable automatic removal of items from the cache when out of memory, and set a large time to live value to prevent automatic eviction.

3.2.6.3 in-Memory Hash Table

Replacing HBase with Memcached reduces the average latency of phrase frequency look-ups to some extent, but each request still goes over the network to query a Memcached server. Carefully analyzing the size of the filtered phrase count dataset reveals that it can fit into one single node’s memory for certain problem sizes. For example, the size of the filtered phrase counts for a sample corpus of 100 MB is 170 MB, which can fit into a node’s memory using a simple hash table. Instead of querying a distributed storage system by sending and receiving data over the network, each mapper can read this data into the memory and query a hash table. If the hash table does not contain an entry for a phrase \( p_j \), a null reference is returned, which is then replaced by 1.

This design alternative has its own cons and pros. As of 2010, a typical compute node in a cluster has 1-2 GB of memory per core. This introduces a limit on the size of the input. Unlike Memcached, we cannot solve the memory problem by simply throwing new nodes because each node creates its own copy of the filtered phrase count data set. On the other hand, storing everything locally in the memory completely eliminates the need for a low latency shared resource, giving rise to a significant performance increase.

3.2.6.4 Utilizing a Bloom Filter

Storing the filtered phrase count data locally in a hash table gives the highest throughput and lowest latency possible, but this solution does not scale. Using distributed storage mechanism like HBase or Memcached are good alternatives in terms of scalability, but higher network latencies associated with each query decrease the performance of the chunking application. If we analyze the query results returned from the storage system along with the phrase count histogram from previous section 3.2.2, we see that approximately 85% of the requests return a null reference. The difference is reviewed in Figure 3.5.
Figure 3.5 – Distribution of the answers returned from the storage system: 85% of the time, the storage system returns a null reference, which is a vast majority because we spend most of the time on receiving nothing but a null reference.

The 85%-15% query result ratio shouldn't be confused with the 98%-2% phrase count ratio discussed previously. Observe that only 2% of the phrase count data includes phrases having count > 1, but we fetch the frequency of those phrases multiple times from the storage system, not only once. For example, the count associated with “of” is fetched thousands of times during execution. That's why 2% of the phrases account for 15% of the overall read requests.

This observation can be put to good use as a more clever optimization method. 85% of the time we issue a read request to the storage system, the query goes through the network, and finally returns a null reference. If we can skip these requests and eliminate the need to wait for a round trip time for each such query, we can give rise to a considerable speed up.

We use a Bloom Filter (20) to reduce the number of unnecessary storage system queries. In a Bloom Filter, false positives are possible but false negatives are not. When inserting a <phrase, count> pair into the storage system, we first mark k bit locations in the Bloom Filter using k deterministic hash functions.

Before querying the storage system to fetch the frequency $c_j$ of a particular phrase $p_j$, the Bloom Filter is queried first. If all of the corresponding k bit locations are set, we further query the storage system. If any of the k bits is not set, there is no need to issue a storage system query because we are 100% confident that the phrase of interest does not have a count greater than one. A false positive happens when all k bit locations are set in the Bloom Filter, but the storage system doesn't contain an entry for $p_j$. In this case, a null reference is returned which implies $c_j = 1$. Figure 3.6 and Figure 3.7 show the basic operation of a Bloom Filter.
Figure 3.6 – Inserting records to a Bloom Filter: For each record, we set three bit locations in the vector. Observe that one of the bit locations is not set for record $d$. This implies the data set does not contain it, so there is no need to issue further queries to the distributed storage system.

Figure 3.7 Using a Bloom Filter to Reduce the Database Accesses: False positives are positive whereas false negatives are not. A Bloom Filter may fail to filter a record, which actually exists in the database. However, a ‘no’ answer definitely implies that particular entry does not exist in the database. The false positive ratio can be customized and it depends on several parameters.

The false positive probability $p$ depends on $m$ the size of the Bloom Filter, $k$ the number of hash functions used, and $n$ the number of items inserted. The parameters for optimal performance are specified as follows.

$$m = -\frac{\ln(p)}{[\ln(2)]^2} \quad \text{and} \quad k = \frac{m}{n}\ln(2)$$
Based on these parameters, we were able to filter 99\% of the unnecessary storage system queries for a 100 MB sized input corpus employing a Bloom Filter of size 6 MB and 7 hash functions. We designed and implemented three different variations of the Bloom Filter mechanism.

In the first two alternatives, we use a Bloom Filter between the client (mapper) and the storage system, which is either HBase or Memcached. Each mapper reads a local copy of the Bloom Filter into the memory and queries it before going further to HBase or Memcached. More than 84\% percent of the time, the Bloom Filter successfully returns a negative answer and the client application doesn't issue a read request to the storage system.

The third alternative has three layers: A Bloom Filter, a local cache, and HBase. When a client wants to fetch the frequency $c_j$ of a phrase $p_j$, it first queries the Bloom Filter. A "no" answer indicates $c_j = 1$, whereas a "yes" answer means $c_j$ can be greater than 1, so the local cache is queried next. The local cache is an in-memory hash table which contains previous entries recently fetched from HBase. We use LRU (least frequently used) as the eviction policy. If the local cache contains an entry for $p_j$, the client doesn't need to issue any database requests to HBase. Otherwise, HBase is queried and the result is returned to the client. The operation of this three-layered storage mechanism is depicted in Figure 3.8.

Figure 3.8 – Operation of the Three-Layered Distributed Storage System: There are four alternatives when the system receives a query request. I) The Bloom Filter returns a ‘no’ answer indicating that further layers do not contain such an entry. This implies the count of the phrase being equal to one. II) The Bloom Filter returns a ‘yes’ answer and the client further queries the local cache. The cache contains the corresponding entry because it was previously fetched from HBase. III) The local cache does not contain such an entry. Client application further queries HBase for the phrase count. HBase returns a value other than null. This value is inserted into the cache to speed up future queries. IV) HBase does not contain an entry for the phrase. This indicates a false positive answer from the Bloom Filter. The phrase count is equal to one, but the Bloom Filter failed to cut down this request. This is the worst-case behavior of the system and it happens rarely, based on the false positive ratio of the Bloom Filter.
Chapter 4

Experiments

4.1 Evaluation and Discussion

We evaluated the performance of our alternative storage systems on the statistical chunking model. We focus on the statistical model because it requires multiple iterations over the input corpus, and our main objective is to minimize the time per one iteration. The input is a 12 MB corpus, which generates 2.6 GB of <phrase, count> pairs as a result of the phrase count application.

All of the 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. For each Map Reduce job, we limited the maximum number of mapper and reducer instances to two per worker node, and speculative execution was disabled. Hadoop by default creates three copies of each data block in the distributed file system to provide fault tolerance. We set this parameter to one, since our cluster is small and we did not want the framework to spend too much time copying blocks between the nodes.

4.1.1 Running time

The purpose of this experiment was to measure the run time performance of the sentence chunking application, and characterize the effects of six different storage mechanisms under heavy load. Our three figures of merit were running time, the average latency per read request and throughput. For each experiment, we performed five runs and reported the arithmetic mean. Figure 4.1 shows the run time of one iteration using several alternative storage mechanisms.
Figure 4.1 – End-To-End running times per one iteration over the corpus: The running times also include the time spent on starting a Hadoop job, reading the input and writing the output via the distributed file system. In our initial implementation utilizing HBase alone, each iteration used to last longer than two hours. We were able to reduce it to as less as two and a half minutes using an in-memory Hash Table. Reducing the execution time of the iterations is significant because the main Hill Climbing algorithm relies on searching the solution space by performing a large number of iterations given the complex and parameter rich nature of the problem.

Not surprisingly, we obtained the best performance using an in memory hash table as the storage system because the read requests stay in the local machines and so there are no issues with network latencies, scanning database files or memory buckets for cache references. There is a dramatic performance difference between HBase and in memory hash table. We were able to reduce the time required for one iteration from more than two hours to less than three minutes, which is more than a factor of 40. This experiment also demonstrates that the main performance bottleneck for the application is the network latency, not the overhead that comes from searching database files for a particular row key. Observe that Memcached was only twice as fast as HBase, although it stores everything in the memory. An in memory hash table achieved 23 times better performance compared to Memcached, and the principle difference between these two systems is serving the queries locally inside each node vs. remotely over the network.

4.1.2 Latency and Throughput

We measured the latency by computing the time difference (in milliseconds) between requesting and receiving a phrase count. Each mapper emits its cumulative latency and the number of read requests as a <key, value> pair, and a reducer aggregates the results, finally dividing the total latency by the total number of read requests to compute the average latency.
Throughput in this context means the average number of memory requests served per second by the storage system, and is computed in a different manner. We implemented a variation of the sentence chunking application, which is exactly the same in functionality with only one difference: It does not issue any storage system requests, but simply uses a constant as the phrase count. This implementation acts as a "No-Op" variant in our throughput experiments, as we measure its end-to-end running time as $T_1$.

When calculating the throughput of a particular storage system, we measure the end-to-end running time of the system as $T_2$ and compute the difference $T_2 - T_1$. This difference is the total time spent by the storage system to issue the read requests. The throughput is then given by $\frac{T_2 - T_1}{n}$, where $n$ is the total number of read requests issued by the mappers.

The latency and throughput performance of the storage systems are summarized in Figure 4.2 and Figure 4.3.

Figure 4.2 – Average latency for a single look-up of a phrase count: The in-Memory Hash Table achieves the smallest latency, which is approximately $10^{-3}$ milliseconds, and thus not visible in this figure. Observe that the addition of a Bloom Filter lowered the latency of HBase and Memcached look-ups from around 0.88 to 0.17 and 0.42 to 0.08 milliseconds respectively. Furthermore, the combination of HBase + Bloom Filter + Local Memory performs better than Memcached + Bloom Filter. This reveals that utilizing the physical memory of the compute nodes locally is more efficient than making use of the physical memory as a distributed Hash Table and dealing with network latencies.
Among the distributed alternatives, the combination of HBase + Bloom Filter + Local Memory results in the highest throughput and it performs approximately 15 times better than using only HBase and relying on the network for every single query. On the other hand, storing all of the filtered phrase counts in each compute node’s memory performs the fastest since each query is served directly from the local memory. The downside of this implementation is its requirement for a compact data set that can be stored in the form of a local Hash Table.

These graphs point out that utilizing a Bloom Filter has a significant performance effect on the latency and throughput. It gives more than five times speed up in Memcached and HBase implementations, which is consistent with the data since we filter 84% of the memory accesses using the Bloom Filter. Another interesting outcome of this experiment is the performance of the three-layered system, which utilizes a local cache in addition to the Bloom Filter. Observe that it gives the smallest latency and highest throughput among the five distributed storage alternatives. This is an expected result because the phrase count data follows a Zipfian distribution, so there are a small number of phrases that occur frequently in the corpus. Once the client fetches the count of such a phrase from HBase and caches it, following requests to the same phrase are served directly from the memory cache.

4.1.3 Accuracy of the Chunking Models

Testing the accuracy of the chunking models is a more challenging task because there is no single answer. A sentence may be divided into smaller partitions in many different ways. We simply used the average number of words per partition as an evaluation criterion. As future work, we are planning to use Amazon Mechanical Turk (41) and compare the results of our models with human judgment.
Our initial experiments with the Heuristic Model resulted in acceptable chunkings. Here is an example output from this model:

The purpose of this study was to investigate the size dependence of the penetration of nanoparticles after local delivery into the vessel wall of the aorta abdominalis of New Zealand white rabbits.

However, we found out that this model is quite sensitive to parameters of the alpha-beta weighting functions (section 3.1.2), and their ranges. A larger input size increases the phrase frequency ranges, so the break-even points of these weighting functions should be moved forward accordingly.

The statistical model is still in development. The normalization factor (section 3.1.1) failed to resolve the issues related to multiplying probabilities in most cases. Our future work will focus on fixing this model using an appropriate normalization factor that involves a more complex analysis over the input corpus.
Chapter 5

Conclusion and Acknowledgement

5.1 Conclusion

This paper deals with partitioning sentences into their constituent terms using two alternative chunking models. The first model is based on a statistical background, and requires multiple iterations to maximize an objective function whereas the second model is built on simple heuristic rules. Implementing both of these algorithms is a challenging task because it requires processing GB’s of text data at each computational step while maintaining an efficient shared data storage system. We have produced methods that allow for the rapid evaluation of our chunking algorithms on parallel computers reducing the computation time by several orders of magnitude.

We adapt our models to the Map Reduce programming framework and build a scalable, time efficient infrastructure to process large amounts of text data in parallel. We make use of state of the art data intensive computing technologies including Hadoop, HBase and Memcached. Our efforts focus on the design and implementation of alternative distributed storage systems used to build a low latency <key, value> look-up table that is accessible from every compute node. We further exploit the Zipfian distribution of the data and utilize a Bloom Filter to minimize the amount of network I/O while maintaining the accuracy of our results. Experimental evaluation proves that a multi layered distributed storage architecture achieves noticeable speed up compared to a naive distributed database system. We argue that minor variations of this framework can be used in similar problem domains such as machine translation which involves the need for a shared <key, value> look-up table for probability distributions.

The statistical sentence chunking model is still under development, and our future work will focus on the accuracy of this model by introducing better normalization factors. Evaluation of the algorithms themselves awaits the availability of index data against which to test them. However, we are already learning some interesting things about sensitivities, normalization methods for statistical methods, and accurate parameter selection.

We will also investigate the performance and scalability of our infrastructure on larger corpora, employing cloud-computing technologies and test our systems on hundreds of compute nodes.
5.2 Acknowledgements

Bibliography


