
Karlsruhe Institute of Technology
Department of Informatics
Institute for Anthropomatics

# Distributed N-Gram Language Models: Application of Large Models to Automatic Speech Recognition

Christian Mandery

Studienarbeit

08.04.2011 – 11.08.2011

Advisor: Prof. Dr. Alex Waibel
Advisor: Dr. Sebastian Stüker

**Abstract**

Language models play an important role in many natural language applications, with n-gram language models representing the most commonly used type for automatic speech recognition and machine translation tasks. Performance of n-gram language models usually increases if a larger text corpus is considered for estimation of the language model or the language model order is raised, which both results in a larger language model. However, the language model size is traditionally limited by the amount of available main memory since the language model must be kept in memory for performance reasons.

In this report, we introduce an architecture that allows to distribute n-gram language models across an arbitrary number of hosts, thus overcoming the memory limitations for such language models. In the presented architecture, n-gram language models are stored distributed across an almost arbitrary number of servers that are queried over the network by clients running a speech recognition system.

We discuss different design aspects of such an architecture like strategies for n-gram distribution or database formats for n-gram storage with their respective pros and cons and provide details on the implementation of important elements of our architecture.

The experimental section of this report contains an empirical evaluation of the discussed n-gram distribution strategies based on a large real-world language model. Timing experiments of the distributed language model show that it is fast enough for application to speech decoding tasks. Finally, we demonstrate that there is an advantage in terms of word error rate (WER) from using larger n-gram language models.

**Acknowledgements**

I would like to show my gratitude to my advisor Dr. Sebastian Stüker for his guidance and advice during this report. By my discussions with him, I learnt a lot about language modelling and speech recognition. Additionally, I would like to thank Prof. Dr. Alex Waibel for arousing my interest in the topic by means of his lectures and Kevin Kilgour who provided valuable help on several occasions.

Furthermore, I am grateful for the permission to perform my computations on the HPC cluster of the Institute for Anthropomatics.

# Contents

# Chapter 1

# Introduction

Language models are widely used in many fields of natural language processing. The basic task of a language model is to assign a probability to an utterance, e.g. a sentence, estimating how likely it is to observe this utterance in the language. As such, the performance of the language model often has a critical impact on the performance of the application as a whole.

The most commonly used approach to language modelling for automatic speech recognition (ASR) and machine translation (MT) tasks is the n-gram approach which is a purely statistical approach of estimating probabilities for new utterances by collecting statistics from a text corpus. For the same language model parameters (smoothing method, pruning, etc.), using a larger text corpus or increasing the model order typically improves language model performance, but also increases the size of the language model. Because most applications require the language model to be kept in main memory, limitations on memory also constrain the size of the text corpus or require other measures to reduce the model size like pruning.

In this report, we present of method of storing and querying n-gram language models for automatic speech recognition distributed across several hosts, implemented as an extension to the Janus Recognition Toolkit. This abolishs the size limits on loading language models and allows for arbitrary large n-gram language models to be used for speech recognition.

In Chapter 2 we will give an introduction to (n-gram) language modelling that explains commonly employed techniques like smoothing, pruning and interpolation and which measurements are used to evaluate language model performance. Then, we will have a look at the used toolkits, namely the SRILM Toolkit that is used to estimate language models and the Janus Recognition Toolkit (JRTk) that provides the speech recognition system that will be extended with the distributed language model. Finally, existing work in the field of distributed language models is discussed.

Chapter 3 explains the basic design behind the distributed system and considers possible strategies that could be used to assign n-grams across several hosts. It will be explained how those language model "parts" are stored in the hosts of our distributed system and what protocol has been implemented for network communication between the client and the distributed servers.

Finally, in Section 4.1, the distribution strategies from the previous chapter will be evaluated in practice. Section 4.2 shows timing experiments that explore the slowdown incurred by the distributed approach. Finally, Section 4.3 elaborates on the central point of quantizing the improvement in speech recognition performance by using larger languages models, as made possible by the distributed system.

# Chapter 2

# Statistical Language Models in Automatic Speech Recognition

This introductory chapter describes the most fundamental basics of statistical language modeling that we will built upon in the following chapters. After a formal definition of a statistical language model, we will have a brief look at the variety of fields where language models play a crucial role. The different measurements that are widely used to estimate the performance of language models will be explained. Hence focussing on n-gram language models, we will discuss different ways of language model estimation, i.e. the process of building a language model based on a text corpus, and introduce the SRILM Toolkit [Stolcke, 2002] and the JANUS Recognition Toolkit (JRTk) [Soltau et al., 2001], both of which are used during this report. Finally, we will discuss the existing efforts on distributed language modeling.

## 2.1 Statistical Language Models

### 2.1.1 Definition

The task of a statistical language model is to judge the probability of a given word sequences, or, more precisely, estimating the conditional probability of observing a specific word in a given linguistic context. From a stochastical point of view, the language model provides a probability distribution over all imaginable sentences.

Let $w_i$ be word $i$ of a word sequence of length $n$ and $W_i^j$ the subsequence from word $i$ to word $j$. The probability of observing the complete word sequence $W_1^n$ can then be estimated from the conditional probabilities of observing the single words in their respective contexts:

$$P(W_1^n) = \prod_{i=1}^{n} P(w_i \mid W_1^{i-1})$$

Typically, additional tokens are inserted as pseudowords into the vocabulary, such as sentence start and end tokens (commonly dubbed "<s>" and "</s>" or "<S>" and "</S>") and an unknown word token that is used to indicate out-of-vocabulary words[1] (commonly dubbed "<unk>" or "<UNK>").

[1]The vocabulary itself can be built dynamically by including all words appearing more often than a given threshold. See [Brants et al., 2007] for an example.

### 2.1.2 Fields of Application

Statistical language models are employed in many different fields of natural language technologies including automatic speech recognition (ASR), statistical machine translation (SMT) or optical character recognition (OCR). In this work, we only used the application to automatic speech recognition to evaluate the performance of the distributed language model. Nevertheless, all other applications of language models (as long as used in the same domain) should benefit accordingly from improved language models that represent natural language more closely.

The system that generates textual hypotheses from acoustic speech input is often called a decoder, a term coined when viewing the speech recognition problem as an information theoretical problem where the information is transmitted over a noisy channel. Besides the language model that determines the probability $P(W)$ for the word sequence $W$, the other important component of such a decoder is the acoustic model that answers the question "how probable is it to observe the input signal, given a specific word sequence?", i.e. $P(X \mid W)$.

To the basic maximization problem of finding the best word sequence $\hat{W}$ in speech recognition as shown in equation 2.1, we can apply Bayes' law to produce equation 2.2.

$$\hat{W} = \underset{W}{\operatorname{argmax}} P(W \mid X) \tag{2.1}$$

$$\hat{W} = \underset{W}{\operatorname{argmax}} \frac{P(X \mid W) \cdot P(W)}{P(X)} \tag{2.2}$$

Because the denominator in 2.2 is the same for all possible interpretations of a given set of input (feature vectors), it can be left out without any effects on $\hat{W}$. This leads us to the fundamental formula of speech recognition:

$$\hat{W} = \underset{W}{\operatorname{argmax}} P(X \mid W) \cdot P(W) \tag{2.3}$$

### 2.1.3 N-gram Language Models

The most prevalent type of a statistical language model is the n-gram language model which makes a markov assumption, assuming that the probability of observing a specific word $w_i$ only depends on the last $n-1$ observed words, i.e. $W_{i-n+1}^{i-1}$. In earlier times, mostly bigram models ($n = 2$) have been used used, whereas nowadays language model orders of $n = 3$ (trigrams), $n = 4$ or even $n = 5$ are common.

Although the simplicity of an n-gram language model obviously cannot possibly convey the complexity of real natural language and research into complexer types of language models has been conducted for decades [Jelinek, 1991], n-gram language models persist as a very popular type of language model used.

### 2.1.4 Smoothing / Backoff

It is worth noting that the size of an n-gram language model grows with larger n-gram orders. Additionally, the problem of data sparsity also exacerbates with increasing orders of the language model, meaning that there will be more n-grams that are perfectly valid parts of natural language albeit they do not appear in the

text corpus. Because the number of possible n-grams grows exponentially with the order, higher n-gram orders require a much larger text corpus for reliable estimation of the language model.

When using a simple maximum likelihood estimation, n-grams that cannot be observed in the corpus would be assigned a zero probability by the language model. Especially in the context of automatic speech recognition, having n-grams assigned a zero probability would be problematic because, should these n-grams, howsoever unlikely, occur in the speech input, they could not be recognized[2]. Additionally, n-grams appearing exactly once (so called singletons) would be assigned an infinitely higher probability than those with no appearance in the text corpus, although empirical experience shows that even removing all singletons entirely from the language model only has a marginal impact on performace [Katz, 1987].

Thus, smoothing techniques are employed to compensate for the sparseness of data and to smooth the distribution obtained by counting n-gram occurences in a limited corpus. As a simple example that performs rather poorly, additive smoothing is accomplished by adding a certain offset $\delta$ to all n-gram counts and use those new pseudocounts for language model estimation. This completely removes the possibility of unseen n-grams and thus reduces the quotient in likelihood estimation between singletons and unseen n-grams, for example from $\infty$ to two for $\delta = 1$.

Better smoothing techniques involve discounting (i.e. reducing) the existing probability values for all seen n-grams. Since the sum of the probabilities for all words in a specific linguistic context needs to sum up to one, the freed probability mass can then be redistributed to account for the occurence probability of unseen words in this linguistic context[3]. This is accomplished by the so called backing-off to (n-1)-gram probabilities for unseen n-grams. Backing-off is done recursively until an n-gram is found in the language model, thus in any case for a known unigram distribution or by assuming uniform distribution at unigram level.

For lack of space, we only want to describe Katz smoothing here as an example of a more sophisticated smoothing technique. For Katz smoothing, n-gram counts $c(W_{i-n+1}^{i})$ are discounted by an n-gram count specific discount factor $d_{c(W_{i-n+1}^{i})}$ which is high for lower n-gram counts and one for n-gram counts above a given threshold, as these n-grams are considered "reliable". Then, $P(w_n \mid W_1^{n-1})$ can be computed as shown in equation 2.4 by using the pseudocounts and applying $\alpha(W_{i-n+1}^{i-1})$ as a normalization factor, chosen such that the sum of all counts given $W_1^{n-1}$ is unchanged and thus conditional probabilites add up to 1 again.

$$P(w_i \mid W_{i-n+1}^{i-1}) = \begin{cases} d_{c(W_{i-n+1}^{i})} \cdot \frac{c(W_{i-n+1}^{i})}{\sum_{w_i} c(W_{i-n+1}^{i-1})}) & \text{if } c(W_{i-n+1}^{i}) > 0 \\ \alpha(W_{i-n+1}^{i-1}) \cdot P(w_i \mid W_{i-n+2}^{i-1}) & \text{if } c(W_{i-n+1}^{i}) = 0 \end{cases} \tag{2.4}$$

[Chen, 1998] considers Modified Kneser-Ney smoothing [James, 2000] to be the most appropriate smoothing technique available for common applications. Modified Kneser-Ney smoothing is based on the interpolated Kneser-Ney smoothing developed in [Kneser and Ney, 1995], but differs in that multiple discounting constants are used instead of one: $D_1$ for n-grams appearing once (singletons), $D_2$ for n-grams

[2] As equation 2.3 shows, the score of a hypothesis is calculated as the product of its acoustic and language model scores, meaning that a language model probability of zero will render the whole hypothesis impossible.

[3] This includes n-grams that have been pruned from the language model to reduce its size.

appearing twice and $D_{3+}$ for n-grams appearing more often. The values for $D_1$, $D_2$ and $D_{3+}$ are calculated from the text corpus from what is known as the count-of-counts, i.e. the total number of n-grams appearing $i$ times in the corpus ($n_i$). As an example, equation 2.5 shows how the discounting for singletons $D_1$ is computed.

$$D_1 = 1 - 2 \cdot \frac{n_2}{n_1 + 2n_2} \tag{2.5}$$

### 2.1.5 Performance Measurement

A measure for the quality of language models involves computing the cross entropy $H(T)$ that indicates how likely a set of test sentences $T = \{t_1, ..., t_n\}$ is, given the language model to be measured. The set of test sentences should be from the same domain as the later application of the language model and must of course not be part of the text corpus used for estimation of the language model.

$$H(T) = -\sum_{i=1}^{n} P(t_i) \cdot \log P_M(t_i)$$

In general, not the $H(T)$ itself is looked at but the perplexity $PP(T)$ which equates to the power of two of the $H(T)$:

$$PP(T) = 2^{H(T)}$$

An natural interpretation of the perplexity is the average branching factor of the language according to the language model [Rosenfeld, 2000]. Therefore, a lower perplexity indicates a "more unambiguous" language model which is generally considered good.[4]

However, using perplexity as a measure for language model performance suffers some severe drawbacks. For example, though increasing the vocabulary size can make a language model better, adding new words not present in the test set will ceteris paribus lower the probability for the sentences in the test set, thus increasing cross entropy and perplexity. Additionally, the difficulty of the branching decision provided by the language model not only depends on the number of possible successor words. For instance, choosing from two similar sounding words can be harder than choosing from a much larger number of words that are clearly distinguishable by the acoustic model.

Fortunately, most of the time it is not a problem to directly measure the influence of the language model on its application. For automatic speech recognition and other applications like optical character recognition, the word error rate (WER) can be determined.

Computing the WER involves generating the most likely hypotheses for sentences from a given test set either by decoding speech or by rescoring sets of given hypotheses in the form of n-best lists or word lattices. Then, the errors made are determined by computing the minimum total error count, whereas errors are categorized as insertions, substitutions and deletions, comparing the hypotheses to manually-crafted reference strings that are deemed "correct". By dividing the total error count by the number of words in the reference transcripts, the WER results.

[4]This can be seen directly by looking at the cross entropy as well: A lower cross entropy indicates that there are smaller deviances between the probability distribution represented by the language model and the "real" distribution implied by the test set.

$$\mathrm{WER}(hypo, ref) = \frac{\mathrm{Ins}(hypo, ref) + \mathrm{Del}(hypo, ref) + \mathrm{Sub}(hypo, ref)}{\mathrm{Count}(ref)}$$

This problem is equivalent to computing the Levenshtein distance at word level and as such efficiently solvable using the principles of dynamic programming.

It is important to remember that the WER does not solely depend on the language model. When hypothesis are calculated as the result of a decoding, they are also influenced by the acoustic model. Otherwise, when they are calculated by rescoring n-best lists or lattices, the word accuracy will be impaired if those are of bad quality. For example, if a word lattice does not contain the correct transcription, even the best language model will never be able to reach a WER of zero. That is why it is better not to examine absolute WERs, but instead differences between WERs computed within the same test environment with all parameters except the language model kept the same.

## 2.2 Language Model Estimation

### 2.2.1 Estimation procedure

Estimation describes the process of training the parameters of a language model on the basis of a text corpus. For estimating back-off n-gram language models, it is therefore necessary to assess probabilities for all {1,..,n}-grams and back-off weights for the {1,..,n-1}-grams contained in the language model.

The first preparatory step aims to improve the quality of the corpus by removing undesirable parts of it, called "cleaning the corpus". For example, parts like special characters, meta data, elements of formatting (e.g. HTML for a corpus collected from the WWW) or sometimes even whole sentences that are considered to be "low quality" are removed[5]. Additionally, transformations like converting numbers to their textual representation might be performed in the cleaning step. Good cleaning is a science in itself that we do not want to focus upon in this report.

In addition to the text corpus, the vocabulary is needed for estimating a language model. It can either be given or it can be dynamicaly determined from the corpus by selecting all words that appear more often than a specified threshold. For the language models that we estimated for the experimental section of this report, a static English vocabulary consisting of 129,135 words is used.

With the corpus and vocabulary given, the actual estimation of the language model parameters is a simple task in principle: First, the {1,..,n}-grams in the text corpus are counted. For this step, the corpus can be split into multiple parts which are counted separately. In doing so, memory requirements can be reduced to fit with (almost) arbitrary low memory constraints and additionally, a speedup can be achieved if the counting tasks are run in parallel. The second step of merging the individual counts files into a single global counts file can be performed without reading all n-gram counts into memory first if the n-grams in the counts files have been sorted, thus reducing main memory requirements by far.

[5] A commonly used criterion is to remove those sentences whose portion of unknown words exceeds a certain threshold.

After the counting step, the language model probabilities are devised from the n-gram counts which is usually done by computing the maximum likelihood probability after applying discounting to the counts to compensate for unseen but possible n-grams (see Subsection 2.1.4). Based on the language model probabilities, the back-off weights are chosen such that the conditional probabilities for every linguistic context sum up to one. As said, [Chen, 1998] gives a good overview about the various existing smoothing approaches and includes an analysis of the impact of smoothing on language model performance.

Although there have been efforts to dynamically grow n-gram language models with some success [Siivola and Pellom, 2005], often this last step of estimating probabilities from n-gram counts is the bottleneck step whose memory requirements limit the total size of the language model that can be estimated.

### 2.2.2 Saving n-gram language models

The common file format for saving n-gram language models is the text-based[6] ARPA back-off language model (ARPABO) format. It is the preferred format for storing n-gram language models in a human readable way and also the format used for interoperability between different involved tools or toolkits. When the same language model needs to be processed multiple times with the same toolkit, however, often using a toolkit-specific dump-style binary format can yield a speedup when loading language models or even reduces temporary memory space needed during the loading process which in turn allows larger language models to be loaded at all.[7]

The ARPABO format starts with a file header that declares how many {1,..,n}-grams are comprised in the language model:

```
\data\
ngram 1=100
ngram 2=10000
ngram 3=200000
```

After the header, one section for each order $m$ contains all the m-grams. It is important to note that the ARPABO format makes no guarantees on the order of the m-grams inside such a section (e.g. sorted alphabetically). Most tools however at least group together those m-grams that share the same linguistic context. This was also assumed as a requirement for the input of the tools written for this report.[8]

In a real-world language model an excerpt from the bigram section could look like this:

```
\2-grams:
...
-2.549887       President Kennedy       -0.1631612
-2.501147       President Reagan        -0.1346047
...
```

[6] Usually language models are stored in gzip-compressed format reducing both filesize and load time on I/O-bound systems.

[7] The Janus Recognition Toolkit (JRTk) mentioned later supports memory-mapped dumps which can be used by multiple running processes without wasting memory for duplication and abolish any loading time by loading needed pages from disk on their first access.

[8] If this assumption is not been met, the state is detected and a warning is printed.

Language model calculations are done using log probabilities (to the base of 10) because addition of the log probabilities equals multiplication of the probabilities but is faster to perform. Saving log probabilities in the first place then removes the need to convert when loading the language model. Therefore, the language model shown above estimates the probability of observing the word "Kennedy" in the context "President" to $10^{-2.549887} \approx 0.28\%$.

The value at the end of the line specifies the logarithmized back-off weight and is thus missing for n-grams of the highest order.[9] In our example, probabilities for the trigram "President Reagan <w>" would be computed as $10^{-0,1346047} \cdot$ P(Reagan <w>) if this trigram is not contained in the language model. Of course, this also means that the probabilities of all existing trigrams with the linguistic context "President Reagan" should sum up to $1 - 10^{-0,1346047} \approx 26.65\%$.

The ARPABO format allows multiple language models to be stored in a single file, which is why after the last n-gram section, a new language model may be specified by starting with a new header. The more common case however is to limit oneself to one language model per file. After the last language model, the end of the ARPABO file is indicated by the end token "\end\".

### 2.2.3 The SRILM Toolkit

The SRI Language Modeling (SRILM) Toolkit [Stolcke, 2002] is a comprehensive and widely used toolkit for estimation and evaluation of numerous types of statistical language models and related tasks. It is being developed since 1995 and free for noncommercial use.

All language models used in this report have been created by SRILM due to SRILM's ability for building large language models by counting n-grams in a concurrent manner (see above) and by saving memory by selectively retaining only those smoothing parameters needed when using a provided helper script called make-big-lm. Still, even with SRILM it is not possible to create language models of arbitrary size, which is why interpolation needs to be used for utmost large language models.

### 2.2.4 Interpolation

Given enough disk space, in theory n-gram language models of arbitrary size can be estimated with little memory due to the fact that every modern operating system supports memory paging. Even so, the estimation process slows down by a factor of several magnitudes if not most of the required data can be hold in memory, making it unbearable slow for practical usage. As a rule of thumb, using the SRILM Toolkit with Modified Kneser-Ney smoothing to estimate a language model that reached 6.7GB in size (gzip'ed) peaked at a memory usage of approximately 100GB during the final estimation step.

Hence, interpolation needs to be used to build language models that are even larger from several smaller language models.[10] A popular approach that performs reasonably well is linear interpolation. It works by interpolation of the n-gram

[9] It is valid to leave it out for lower-order n-grams as well which implies a back-off weight of zero, meaning that no probabilities mass has been redistributed to unseen n-grams for this linguistic context.

[10] Sometimes, this approach is termed static interpolation, in contrast to dynamic interpolation which loads multiple language models at evaluation time and performs interpolation on-the-fly.

probabilities between an arbitrary number of language models using interpolations weights that are calculated by minimizing the test set perplexity[11] of the interpolated language model. Determing those optimal interpolation weights is done iteratively from the perplexities of the initial language models. In a second step, the back-off weights are recomputed based on the new interpolated probabilities.

When using the SRILM Toolkit, there is a helper script called compute-best-mix that automates the described process of finding the best interpolation weights iteratively and (static) interpolation is configured by using the -lm, -lambda, -mix-lm2, -mix-lambda2, -mix-lm3, -mix-lambda3, etc. switches to the ngram tool.

Beside offering a convenient way of building large language models, interpolation is also suitable for generating language models optimized for a specific domain. An imaginary scenario might involve a huge generic language model and several smaller domain-specific language models. A test set from the target domain is then used to estimate the optimal interpolation weights.

### 2.2.5 Pruning

Pruning denotes the process of removing n-grams from an n-gram language model to reduce its size while degrading language model performance as little as possible. The basic idea behind this is to include all available text data in the estimation process as long as enough memory is available, and then keep removing the most insignificant n-grams until the desired language model size is reached[12]. [Seymore and Rosenfeld, 1996] shows that this approach is superior to the alternative of starting off with a narrowed text corpus in the first place.

However, a decision criterion is needed for choosing which n-gram should be pruned first, or more specifically, determining how big the influence of a certain n-gram is on the performance of the language model. The cutoff method uses a simple heuristic for this purpose that removes n-grams based on their occurrence count in the text corpus, pruning infrequently appearing n-grams first[13] The weighted difference method takes into consideration that removing an n-gram from a language model causes the probability of this n-gram to be computed by backing-off. Therefore, the difference between the existing language model probability and the probability by backing-off is considered and n-grams with a small difference are pruned first.

[Stolcke, 2000] resumes work on the problem of selecting n-grams for pruning and presents a method to explicitly determine the effect of a single n-gram on the perplexity of a language model. Afterwards, all n-grams are removed that influence the perplexity less than a threshold value which has to be specified according to the intended size of the pruned language model.

Apart from pruning, there are other approaches like clustering that also aim to reduce the complexity of an n-gram language model. Clustering works by replacing

[11] As usual, this test set should share the same domain as the desired later application of the language model and must not be part of the text corpus used for estimation.

[12] Of course, after n-grams have been removed from a back-off n-gram language model, the back-off weights need to be recomputed.

[13] [Seymore and Rosenfeld, 1996] demonstrates that even with this simple pruning technique, the language model size can be reduced significantly without a relevant effect on the word error rate. This corresponds to the already mentioned observation that singleton n-grams can be pruned with very little performance implications.

"related words" like surnames, color names, month names etc. with a token designating the respective word class, which may even increasing model performance since every language model is estimated on a finite text corpus. An introduction to clustering and other common language model techniques like skip n-gram language models or cache language models can be found in [Goodman, 2001].

## 2.3 Janus Recognition Toolkit

The Janus Recognition Toolkit (JRTk) is a toolkit written in C that has been developed by the Interactive Systems Laboratories (ISL) at the Karlsruhe Institute of Technology (KIT) and the Carnegie Mellon University (CMU). It provides Tcl interfaces for the rapid development of automatic speech recognition applications. For example, it has been used in the JANUS-III speech-to-speech translation system [Soltau et al., 2001]. In the following, "Janus" always refers to the Janus Recognition Toolkit.

Janus employs a state-of-the-art one-pass decoder that utilizes all available information sources, including the full language model, in a single pass. Such a single pass approach eliminates the risk that proper hypotheses are pruned in an early pass and thus cannot be found later on. In addition, it also makes for great speedups compared to a multi-pass approach. [Soltau et al., 2001]

Different types of linguistic knowledge sources are supported by Janus. Besides traditional n-gram language models, phrase language models built on top of an n-gram language model and context-free grammar language models can be used, and multiple language models can be loaded and interpolated on-the-fly.

Janus can perform two different processing steps when loading the logarithmized parameters (probabilities and back-off weights) from an n-gram language model. First, the loaded values can be "compressed" by saving them in a 16-bit integer format (signed short int) instead of the 32-bit single precision floating-point format (float). The compressed value is calculated as

$$\text{compressed} = \lfloor \text{uncompressed} \cdot 1000 - 0.5 \rfloor$$ [14]

By default, compression is enabled and therefore no further floating-point arithmetic needs to be performed after loading the language model.

"Quantization" is a second processing step taking place after the compression. Quantization further reduces the memory demand to one byte per value by saving not the value itself but its index in a generated lookup table. Such a separate lookup table is generated for every set of values $V$ (i.e. "3-grams back-off weights") by determining the minimum and maximum value present in the set of values to establish the overall range of values that must be covered by the lookup table. Hence, the scaling factor $s$ is calculated as

$$s = \frac{256}{\max(V) - \min(V) + 1}$$

Lookup table indices can then be easily computed, as well as the lookup table itself as the inverse function:

[14] As a result, only zero and the negative range of the signed short int are used (log10 scores are always zero or negative).

$$\text{index}(v) = s \cdot (v - \min(V))$$

$$\text{value}(i) = \min(V) + \lfloor \frac{0.5}{s} \rfloor + i \cdot \frac{\max(V) - \min(V) + 1}{256}$$

By default, quantization is only performed for n-grams of the highest order due to the larger range of values for lower-order n-grams and the fact that for typical language models, the biggest share of memory is allotted to the highest-order n-gram section anyway.

Both compression and quantization are implemented to minimize the memory footprint and compression additionally helps to avoid slow floating-point arithmetic. Their influence on language model performance is very slight.[15] Still, in the context of this report it is crucial to pay attention to those processing steps in Janus in order to reproduce them in the distributed language model in exactly the same way. Then, we can validate the distributed language model simply by matching distributed with local scores (or by matching hypotheses).

## 2.4 Existing Approaches to Distributed Language Models

We want to refer to two different approaches to distributed language models for application of large n-gram language models here.

The first approach has been chosen in the papers of [Zhang et al., 2006] and [Emami et al., 2007] in a nearly identical fashion. For this approach, no language model is built at all. Instead the text corpus itself is distributed across a number of servers with each hosting a fraction of the corpus. All servers are then queried by the client running the language model application, e.g. a speech decoder, to gather all needed n-gram counts to estimate the desired language model probabilities on-the-fly. Efficient calculation of the n-gram counts on the servers is made possible by indexing the respective part of the corpus in a suffix array to allow for lookups of occurrences of a word sequence of length $n$ in a corpus of size $c$ in $O(n \cdot \log c)$.

An advantage of this approach is that the size of the text corpus used is only limited by the number of servers available and their memory capacity. Hence, text corpora of almost arbitrary size can be applied. Furthermore, corpora can be added or removed at runtime and several different corpora can be distributed with their weights dynamically altered by the client.

However, a drawback of this architecture is its slowness. Not only is the suffix array lookup significantly slower than merely reading precomputed probabilities from memory, but also, for each n-gram count that needs to be determined, every single server needs to be queried, making the process increasingly slow for larger corpora. With linear interpolation, up to $n$ counts must be retrieved from every server to compute an n-gram probability and for those smoothing methods that empirically exhibit the best performance for local language models (e.g. Modified Kneser-Ney smoothing), even more computation would be required at runtime.

[15] Interestingly, there is anecdotal evidence that quantization actually improves the word error rate in some cases, despite probabilities being mapped to pretty coarse steps.

Another approach has been introduced by [Brants et al., 2007] in which the estimation of an n-gram language model from a large text corpus is facilitated by using Google's MapReduce framework on a large cluster. Afterwards, the distributed language model is again hosted on a number of servers and applied to a machine translation task for performance evaluation. To decrease the overhead by network latency, the decoder has been modified to batch together multiple scoring requests from tentatively extending the search graph on several nodes, rather than retrieving only one score at a given time.

The authors experimented with Kneser-Ney smoothing but found it too intricate to build very large language models in a distributed way. Instead, they present a novel way of backing-off referred to as "stupid back-off" which is extremely simple due to the fact that it does not perform any discounting and uses a single static back-off factor for n-grams of all lengths.[16] Still, it is observed that stupid back-off works reasonably well for very large text corpora and even outperforms the largest sensibly producible language models with Kneser-Ney smoothing when used on outermost vast corpora.

This approach is in many ways comparable to our work presented here. Nevertheless, there are important differences. [Brants et al., 2007] utilize a distributed architecture for building the language model whereas we convert existing language models into a format well-suited for distributed retrieval. Thus, with the approach presented here, all smoothing techniques and existing language model operations like interpolation, pruning etc. that are already implemented by SRILM and other tools can be readily applied to the language model. As a drawback, however, the estimation process becomes the bottleneck that limits the maximum language model size in our approach. Other differences concern the task that is used to evaluate language model performance in that this report uses an automatic speech recognition instead of a statistical machine translation task.

[16] Because without discounting no probability mass is freed but back-off factors are still non-zero, this causes the conditional probabilities to sum up to values larger than one. However, this does not pose a problem in the context of automatic speech recognition or statistical machine translation, since essentially the only question that the language model must answer for these tasks is "which hypothesis is most probable?" but not "how probable is hypothesis X?".

# Chapter 3

# Distributing N-Gram Language Models

In this chapter, we will present the distributed n-gram language model architecture that we devised within the framework of this report. First, we will provide a basic overview of the design that explains which components exist and how they work together. Then different strategies to distribute n-grams across several server instances will be discussed and we will present our simulation framework that allows to assess the various strategies for the distribution of existing language models. In the third section, we will outline the database format that is used by the server part of the distributed system to store the language model parameters and we will describe the process of building these databases from a language model in the ARPABO format. Finally, we will elaborate on the implementation details of both the client and the server part.

## 3.1 Architectural Design

### 3.1.1 Outline

Our approach for the distributed language model uses a conventional client-server architecture in which a server instance is started by providing it with a database file and possibly other options. Henceforward, the server instance is running and waiting for clients to connect to serve their requests from the loaded database. In general, "client" in this relation describes any application that needs to retrieve n-gram probabilities from the server(s).

In the context of this report, the client is implemented as an extension to the regular Janus NGramLM type that represents an n-gram language model. Its Tcl interface has been enhanced with commands to establish (and close) connections to a set of remote servers. As long as the NGramLM instance is connected to remote servers, all subsequently needed n-gram scores are determined by asking one of those servers if the desired n-gram is outside of the scope of the local language model[1]. Because the interface has not been changed otherwise, existing application using Janus can easily be upgraded to use a distributed language model by replacing the command to load the full language model locally with commands to load only a

[1]The local language model may just contain the unigram table, or higher-order n-grams like bigrams as well to increase performace.

lower-order fraction of the language model (in the example, only unigrams) and to connect to a set of remote servers (in the example, two) hosting the remaining n-grams:

```
lmObj load languageModel.arpabo -maxLoadOrder 1
lmObj.NGramLM connectDistribServer "{host1 port1} {host2 port2}"
```

Besides the client, the server has been implemented in C and within Janus, too. Nevertheless, it has very few dependencies to the other Janus modules aside from using the Tcl interface and could be implemented as a stand-alone program as well. For example, it does not share the language model data structures or loading code existing in NGramLM, but utilizes its own functions optimized for memory-efficient conversion of language model files to its database format instead.

A server is created by instantiating an object of the NGramLMDS ("n-gram language model distributed server") type in Janus' Tcl interface. Different methods allow to configure the server, load a database and start or stop the actual server daemon. For matters of namespace tidiness, the code to build server databases from n-gram language models in the ARPABO format is also part of NGramLMDS, despite not really being assigned to a concrete server instance.

Communication between a client and a server is accomplished via a TCP connection. After connecting, the client can send n-grams which are sequentially answered by the server on the same channel. No words in textual form are transferred, but instead, only word IDs are used.[2] Each answer from the server is either the information that the sent n-gram is not contained in the language model, or its probability and possibly its back-off weight. Corresponding to the description in Section 2.3, values sent by server are always dequantized but never decompressed.

Two different protocols have been implemented which both support two query types. While the first query type is the usual request for probability and possibly back-off weight of a single n-gram, the second query type allows for the case of array scoring when the probabilities and possibly back-off weights are needed for all vocabulary words in a given linguistic context. As it will be shown later in the description of the client implementation, this query type is of particular interest in speech recognition when building the search graph, i.e. during decoding.

### 3.1.2 Protocol v1 (ASCII-based)

The first protocol (v1) is an ASCII-based protocol that terminates the separate requests and responses by new line characters[3].

A request for a single n-gram is simply made by sending a white-space separated line containing the word IDs. The response of the server then consists of "notfound" if the n-gram does not exist in the language model or the probability and, if existing, the back-off weight separated by whitespace. A request for an array lookup is made by prefixing the word ID list with the character "a" and setting the last word ID of the n-gram (i.e. the current word) to the maximum word ID which equals to the vocabulary size minus one. Then, for every word known, a response line is returned

[2]Actually, the server has no way to find out which word a word ID corresponds to after its database has been built.

[3]Arbitrary combinations of the carriage return and line feed characters are supported.

analogous to the response to a single-word request which obviously greatly strains bandwidth.[4]

Server-side errors are reported by returning "error/reason" as the response with "reason" specifying the cause of the error. For example, "error/db" indicates a database error, "error/request" a malformed request by the client and "error/order" a request for an n-gram order that is not available in the database.

We have chosen an ASCII-based protocol for the first implementation of the distributed language model architecture to increase comprehensibility and facilitate the debugging process during development.[5] Especially testing the server is simplified since a client can be emulated manually by using a telnet client. Unfortunately, using an ASCII-based protocol has the drawback that all numbers must be converted to text before sending and converted back after receipt which brings a significant overhead to the communication, and additionally, that network traffic is higher.

### 3.1.3 Protocol v2 (binary)

To avoid the overhead from the protocol v1, a second protocol (v2) has been implemented that is binary-based. Requests and responses are not separated by a special marker here which would be intricate for binary data anyway but prefixed by a header that indicates how many bytes of data are following.

A request in protocol v2 starts with a byte, treated as a signed integer, that specifies the number of 32-bit word IDs to follow. If that word ID count is zero or negative, the request demands an array lookup with the word IDs as the linguistic context, so that this lookup concerns n-grams of the order of the word ID count plus one.[6]

The server then responds with a one-byte status code, again treated as a signed integer, that is zero on success or negative on error. For single n-gram requests, the status can also take the value of one to indicate that the desired n-gram could not be found in the database. With this information, the client already knows the response length for a single n-gram request since back-off weights are provided for every n-gram order except the highest order available and the client is aware of the language model order. For array lookups, the number of n-grams found is appended to the status code as an unsigned 32-bit integer. Then, for each found n-gram, the 32-bit word ID is sent, followed by the probability and possibly the back-off weight. Again, the client knows whether it has to expect back-off weights in the response and can thus compute the length of a single n-gram entry for the array lookup.

Comprehensive timing results for both protocols can be found in the experimental section of this report. Though, because both protocols not only differ in being ASCII-based or binary protocols but also in the algorithms used for database

[4] As a small optimization, "notmore" is returned if all following words are "notfound". Still, this bandwidth-wasting behaviour is the presumable reason for the disastrous performance of protocol v1 for array lookups.

[5] One reason why an ASCII-based protocol is easy to debug is that debugging output can be inserted into the client and/or server to dump the whole network traffic in human-readable form without the necessity of further error-prone conversion. But still without such debug output, it turned out that not even a packet sniffer is needed but using strace is sufficient in most cases to grasp what is going on between the client and the server.

[6] Thus, if the word ID count of a request is zero, the server will handle the request as an array lookup with an zero-length linguistic context and reply with the unigram probability and back-off weights table.

lookups in the backend, timing deviations can only serve as an rough indicator for the performance benefit of protocol v2.

## 3.2 Distribution Strategies

### 3.2.1 Introduction

When developing distributed storage systems, it is a crucical question which piece of information is located on which component of the system. In the context of this work, a rule must be established to determine on which server a certain n-gram should be stored. Furthermore, this rule should depend only on information that is available at both the time the n-grams are distributed to the databases of the individual servers and at runtime of the client to ensure that a single server access is sufficient for the client by all means. Because of this, the n-gram words respectively the corresponding word IDs are the main parameters for distribution.[7]

Aside from the distribution of n-grams, the question of redundancy arises. Can information access in the distributed storage system be made more efficient by spreading out certain pieces of information to more than one component? For back-off n-gram language models, backing-off could be done server-side without further network communication if the server had all the n-grams available that are needed to back-off from all imaginable n-grams that would be distributed to this server if they existed. Unfortunately, to date no distribution is known to readily accomplish this [Brants et al., 2007][8]. Therefore and to keep memory requirements as tight as possible, redundancy in n-gram distribution is not further evaluated in this report.

### 3.2.2 Evaluated Strategies

In the following, $k$ refers to the number of servers that constitute the distributed system and $w_1, ..., w_n$ to the numeric word IDs of the n-gram to distribute, with $w_1, ..., w_{n-1}$ representing the linguistic context and $w_n$ the current word.

A valid strategy $\sigma$ assigns every n-gram to a nonempty subset of the available servers. Because redundancy is not further considered in this report, every n-gram is assigned to exactly one server for the strategies discussed here.

$$\forall(w_1, ..., w_n) : \sigma(w_1, ..., w_n) \in \{0, ..., k-1\}$$

A good strategy distributes the available n-grams as uniformly as possible, that is to say that every server should be assigned roughly the same amount of data. Additionally, requests from a client should also be distributed equally across the servers. Most of the time, this follows readily if the n-grams are distributed well. Exceptions occur if a strategy considers the n-gram order and treats n-grams of different orders differently.[9]

[7] For example, the position of an n-gram in the language model file in the form of "lineNumber mod serverCount" is not suitable because this file is not available to the client at runtime.

[8] Apart from the unfeasible solution of distributing all n-grams below the highest order to every server.

[9] For instance, when a trigram language model would have all its unigrams and bigrams allotted to one server and its trigrams distributed among five other servers, the load on the unigram/bigram host will be several times larger than those on one of the trigram hosts although the database size for each host might be roughly the same.

The inclusion of $w_n$ in distribution strategies deserves another thought. As mentioned before, a common request to the distributed language model is the retrieval of the probabilities for all vocabulary words in a given linguistic context. Because of this, array scoring has been implemented that allows to get all those probabilities from a server in a single request. While it would nevertheless still be possible to include $w_n$ in the strategy[10], it would require the client to ask multiple servers even before backing-off. Hence, no strategies that consider $w_n$ are taken into further consideration here.

A elementary strategy uses just a single word index $i \in \{1, ..., n-1\}$ and distributes n-grams uniformly across the servers based on $w_i$:

$$\sigma_{singleword}(w_1, ..., w_n) = w_i \mod k$$

However, the drawback of this *single word strategy* is that the likelihood of the different words is not even close to equality. In fact, some words, including the sentence start and end tokens and the unknown word token, appear far more often than most other words. With the *single word strategy*, this leads to a very unbalanced distribution of n-grams, a problem that grows with larger values of $k$.

Since the largest amount of data assigned to a single server must be smaller than the available memory on a server, having some exceptionally large n-gram fractions requires $k$ to be raised even further, even if all other n-gram fractions would fit into the server memory. In an extreme example, it is imaginable that a suitable distribution cannot be established at all because the n-gram fraction containing all n-grams with a very common word (e.g. "<unk>") as $w_i$ is even for $k = vocabsize$ still too large to be loaded on a server host.

In a more advanced approach, a hash algorithm is used with a subset of the words $S \subseteq \{w_1, ..., w_{n-1}\}$ and the resulting hash value is assigned to a server as before:

$$\sigma_{hash-S}(w_1, ..., w_n) = hash(S) \mod k$$

This strategy reliably provides a good distribution if $|S| > 1$. When hashing with $|S| = 1$, it is obvious that the same problems like with the *single word strategy* can be observed. Hashing the whole n-gram should generally present the best possible distribution strategy if a reasonably good hash function is being used. However, hashing is a computationally intensive operation and it is necessary to ensure that the hashing function of the client is consistent with the one used when building the databases for the servers[11]. Could a more efficient strategy offer a distribution quality close to those of hashing but without the complexity of hashing?

Such a simple strategy is an extension of the *single word strategy* concerning that instead of a single word, the sum of the word IDs in a subset $S \subseteq \{w_1, ..., w_{n-1}\}$ is considered:

$$\sigma_{sum-S}(w_1, ..., w_n) = \sum_{s \in S} s \mod k$$

[10]The client can ask multiple servers and then merge the responses together by overwriting "not found" responses with the score from another server that has this particular n-gram.

[11]This may seem to be a negligible issue, however it can become a problem if multiple programming languages (C, Tcl, Python, etc.) are involved that use different data representations and libraries for hashing.

This *sum strategy* is still almost as simple as the *single word strategy* while avoiding its major drawback if $S$ is chosen large enough. In fact, there are no reasons against using the whole linguistic context for computing the sum.

A theoretic disadvantage of the *sum strategy* is that the server that an n-gram is assigned to is independent of word order. For example, "<unk><unk><s><unk>" will be assigned to the same server as "<unk><s><unk><unk>" when using $S = \{w_1, w_2, w_3\}$. We do not think that this poses a problem for usual language models but nevertheless evaluated another approach. The *weighted sum strategy* resembles the *sum strategy* but differs in that the words are given different weights within the sum. This is achieved by multiplying the partial sum with a constant factor $c$ before each addition and thus produces:

$$\sigma_{weightedSum-S}(w_1, ..., w_n) = \sum_{i=0, s \in S}^{|S|-1} (s \cdot c^i) \mod k$$

### 3.2.3 Evaluation of Strategies with a Simulator

We wrote a simulation tool in C++ to evaluate the discussed distribution strategies. A good strategy should yield a smooth and uniform distribution for the n-grams of a language model, since the minimum amount of memory needed for a single server in a homogenous server environment corresponds to the maximum of the sizes of all built server databases. To evaluate the distribution of a language model, our simulator can read language models in the ARPABO format and simulate different distribution strategies with multiple parameter configurations in parallel.

As a simplifying assumption, n-grams of all orders are counted together, which assumes that e.g. a bigram requires as much space as a trigram, and no strategies have been evaluated that consider the n-gram order for the distribution. Such an asymmetric distribution strategy is not suitable for a homogenous server environment in any case, since the probability that a specific lower-order n-gram is required is generally higher than the retrieval probability for a specific higher-order n-gram, and hence unequal load would be put upon the servers.

Typically, n-grams of all orders up to the language model order are processed by the simulator. Therefore, another thing to pay attention to is the handling of $S$ in the simulator. The word indices in $S$ can be specified from the start of the n-gram ($w_1$, $w_2$, etc.) or from the end ($w_k$, $w_{k-1}$, etc.). For example, $S_1 = \{w_2, w_3\}$ and $S_2 = \{w_{k-2}, w_{k-1}\}$ lead to the same strategy when applied to a 4-gram, but when applied to a trigram, $S_1$ considers the words $w_2$ and $w_3$ whereas $S_2$ uses the words $w_1$ and $w_2$. In addition, elements of $S$ that are "out of range" are ignored in the computation: For bigrams, $S_1$ and $S_2$ consider only $w_2$ respectively $w_1$, and all unigrams are assigned to the same server for both $S_1$ and $S_2$.

The output of the simulator are the numbers of n-grams assigned to each of the servers for all evaluated strategies. To make the results easier to interpret, we compute Gini coefficients for each pair of strategy and number of servers $k$.

The Gini coefficient is a measure developed by Corrado Gini to summarize the degree of inequality of a statistical distribution in a single number. It is defined as the difference between the area under the cumulative distribution function of the distribution to be evaluated and the cumulative distribution function of the standard uniform distribution (i.e. the identity function $f(x) = x$ for $x \in [0; 1]$), normalized

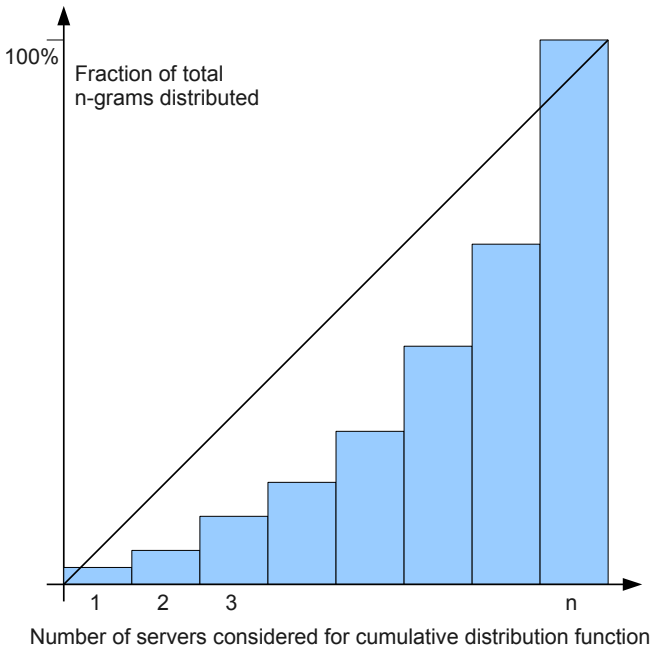


Figure 3.1: Exemplary cumulative numbers of n-grams for every quantile

by the area under the cumulative distribution function of the standard uniform distribution. Figure 3.1 shows an example for the discrete cumulative function of an n-gram distribution that is compared to the cumulative distribution function of the standard uniform distribution, shown in black.

From the definition, we can see that a Gini coefficient of zero characterizes a perfectly uniform distribution, whereas a distribution strategy that assigns all n-grams to one of infinitely many servers would have the maximum possible Gini coefficient of one. Because n-gram counts cannot be negative, Gini coefficients are always less or equal to one, and because the cumulative distribution function is always convex, Gini coefficients are positive or zero.

For the discrete distribution provided by the simulation with a finite number of servers, the integral required for calculation of the area under the cumulative distribution function is replaced by a sum. Therefore, the Gini coefficient $G$ for $k$ servers with their respective n-gram counts $\text{count}(x), x \in [1, k]$ is computed by:

$$G = 1 - \frac{1}{k} \cdot \left( \frac{2 \cdot \sum_{i=1}^{k-1} \text{cumul}(i)}{\text{cumul}(k)} + 1 \right) \quad \text{with cumul}(x) = \sum_{i=1}^{x} \text{count}(x)$$

It should be noted that Gini coefficients are comparable only between simulations using the same number of servers, as the calculation depends on the number of quantiles used for specification of the distribution.

The *sum strategy* with $S = \{w_1, ..., w_{n-1}\}$ has proven to work reasonably well in our experiments and is hence used as the distribution strategy for our distributed language model. The discussion of all simulator results and other interesting findings from the simulator runs can be found in Section 4.1 and the n-gram counts for some

lower values of $k$ and all computed Gini coefficients are available in Appendix A.

## 3.3 Storage Structure

### 3.3.1 Introduction

The Berkeley DB library [Olson et al., 1999] is used in the distributed language model to store all data used by a single server instance. The database for a server must be generated only once from the language model which offers a significant performance advantage over reading the full language model every time a server is started. By default, when a server is started, enough cache space is allocated to comprise the whole database. Then, all database pages are preread from disk so that no more disk accesses are necessary during runtime.[12] For testing purposes, prereading can be disabled or smaller cache sizes be used.

### 3.3.2 Storing n-grams

We have implemented two different ways of storing n-grams. For the first method, every n-gram is stored as its own database record and indexed by using all n-gram word IDs as the key, producing a key size of $4 \cdot n$ bytes. We call this a context+word-indexed database. The value saved in one of those database records only consists of the probability score of the n-gram (1-4 bytes[13]) and, if existing, its back-off weight (1-4 bytes). Obviously, this approach is most suitable for the lookup of single n-grams. It shows very poor performance if used for array lookups, because every imaginable n-gram must be searched independently.

The second possibility is to store all n-grams sharing the same linguistic context together in one database record. The database key for this record then consists only of the word IDs of the linguistic context and therefore, we call the database a context-indexed database. The value of a record in such a database contains a list of all words that exist as an n-gram with this linguistic context in the language model, with each entry consisting of the word ID (4 bytes), the probability score (1-4 bytes) and possibly the back-off weight (1-4 bytes). The advantage of this approach is that array lookups can be easily implemented using a single database access. However, performing a lookup for a specific n-gram is more time-consuming than with a context+word-indexed database since a possibly very large data portion must be fetched from the database and scanned for occurence of the desired word.

The differences in the space requirements of the two approaches are hard to predict. In general, context+word-indexed databases favor language models that contain many different linguistic contexts with few n-grams each, while context-indexed databases favor language models that have many n-grams assigned to the same linguistic context.

| # | Name | Description |
|---|---|---|
| 1. | DB format | Version of the database structure used in the database, allows detection of incompatible older formats |
| 2. | Key format | Indexing scheme used in database, i.e. either context-based or context+word-based indexing |
| 3. | Vocabulary size | Equals the maximum word ID plus one, only necessary for request validation and for array lookups in context+word-indexed databases |

Table 3.1: Structure of the metadata header

| # | Name | Description |
|---|---|---|
| 1. | N-gram order | Order of the n-gram section |
| 2. | Back-offs? | Whether the section contains back-off weights, typically true for all but the highest-order section |
| 3. | Quantized? | Whether scores in this section are quantized and the lookup tables should be used, typically only true for the highest-order section |
| 4. | N-gram count | Number of n-grams in this section, only necessary for informational output |
| 5. | Lookup table for probability quantization | Either 512 or 1024 bytes, depending on whether compression is enabled |
| 5. | Lookup table for back-off weight quantization | Either 512 or 1024 bytes, depending on whether compression is enabled |

Table 3.2: Structure of a section metadata block

### 3.3.3 Metadata

Besides the actual n-grams, additional metadata needs to be stored to provide information about the database format and the language model. This metadata is saved in the database under the one-byte key "m" to avoid possible clashes with the keys from the data records which have a multiple of four bytes as length[14]. The structure of the metadata header that starts the chunk of metadata is shown in Table 3.1. It is followed by an (almost) arbitrary number of section metadata blocks, specified in Table 3.2. Each of these section metadata blocks contains information about all n-grams with a certain n-gram order that are present in the database. Because Berkeley DB returns the size of the value portion when reading the metadata record, there is no need to save the number of sections in the metadata header.

Compression of 32-bit float scores to 16-bit integer scores is not controlled by the metadata but configured as a compile-time option in Janus (enabled by default) and as such respected by the database code. In addition, the option for quantization as described in Section 2.3 is provided and requires the lookup tables to be saved in the respective section metadata block[15].

### 3.3.4 Berkeley DB access methods

Different types of indexes are available in Berkeley DB as what is called the "access method". For the distributed language model, the B+tree access method and the extended linear hashing access method can be chosen when building the database. Because the usage of the database in Berkeley DB is (almost) independent of the used access method, this could be implemented without considerable extra effort. The performance of both access methods is measured and compared in Section 4.2.

### 3.3.5 Building databases

Databases are built from n-gram language models in the ARPABO format. Apart from the filenames of language model and database, this requires a list of sections to be converted (e.g. all n-grams, or only 4-grams), the total number of servers involved and the number of the server to create the database for[16]. The default values for the access method and the key format can be optionally overwritten.

Building the database starts by reading the unigram section of the language model. This is needed to establish a mapping of words to word IDs that is used for the rest of the procedure. Afterwards, each section that should be included is processed separately. Sections that should be quantized require the minimum and maximum value to be known for quantization, which is why a temporary database is created for those sections to store the compressed but not yet quantized n-grams.

[12]The linear reading pattern exhibited by the prereading performs by far better than highly random access pattern during database use.

[13]The actual size depends on the settings for compression and quantization.

[14]A zero-length key is possible with Berkeley DB and actually used for storing the unigram table in context-indexed databases.

[15]Actually, it would be sufficient to save the minimum and maximum values and recompute the lookup tables when loading the database. We still favor saving the lookup tables in the database to preserve the chance to implement a possibly better non-linear quantization technique without changing the database format.

[16]A batch mode allows to build the databases for all of the servers with a single invocation.

Accordingly, a finalization step then quantizes those entries and writes the results to the real database. After all desired sections have been processed, the database is compacted to minimize space requirements for later usage.

The amount of cache memory used by Berkeley DB for the build process is configurable and temporary databases are created as in-memory databases which are paged out to the disk only if the cache memory is exhausted. Thus, if enough memory is available, there is no I/O overhead apart from reading the language model and saving the final database. However, databases can also be built with very little memory, independent of the size of the language model.

## 3.4 Implementation

### 3.4.1 Client implementation

The client for the distributed n-gram language model has been implemented as an extension of the standard n-gram language model in Janus. Since the reason for choosing a distributed approach was to facilitate evaluation of n-gram language models that are too large to fit in the working memory of a single machine, the language model loading code has been modified to allow for skipping n-gram sections above a configurable threshold order. For instance, for a huge 5-gram language model, unigrams to trigrams might perhaps best kept locally, while 4-grams and 5-grams should be distributed because they account for the lion's share of memory consumption. In any case, the unigrams need to be available locally to establish the client-side mapping from words to word IDs.

The implementation of array lookups differs between protocol v1 and v2. Protocol v2 uses an algorithm that resembles the recursive algorithm used for local array scoring and realizes an array lookup for a linguistic context of the non-zero length $k$ by leaving out $w_1$ and performing an array lookup for the context length $k-1$. In the resulting probability array, the entries of those (k+1)-grams that are contained in the language model are then overwritten with their respective values. For all other array entries, the backing-off is accomplished by adding[17] the back-off weight determined from the k-gram representing the linguistic context. The recursion terminates with returning the probability scores and back-off weights for all vocabulary words from the unigram table as soon as $k = 0$ is reached.

Because the client provides the option to skip higher-order n-gram sections when reading the language model, it is capable of loading the same language model in the ARPABO format that has been used to build the databases for the servers. Hence for every scoring request after connecting to the remote servers, the distributed language model is consulted first of all, but the implementation switches to using the locally available data once a sufficiently low n-gram order is reached during backing-off.

Additionally, a verification mode can be used for the purpose of validation. It requires the full language model to be loaded locally but the client is still connected to the remote servers. Afterwards, every scoring request is executed both by means of the local language model and the distributed system. If any discrepancies between the two computed probabilities are detected, the causative n-gram is printed to allow for further investigation. This facilitates debugging of the distributed language

[17] Since all calculations are performed with log probabilities, this equals a multiplication.

model because errors can be detected as early as possible and not only on the application level by comparing word lattices or hypotheses.

### 3.4.2 Server Implementation

The implementation of the server uses a multithreaded architecture to guarantee fair scheduling between multiple connected clients and separate the running server instance from the main Janus thread providing the Tcl interpreter. We favored the usage of Native POSIX Linux Threads (NPTL) over the traditional fork() system call because they have a lighter memory footprint and thus continue to replace fork() in modern applications. After a server instance is started, a thread is launched to listen for inbound connections on the network socket. For each connecting client, a separate thread is then created that handles only requests on this specific connection and makes all necessary database queries on its own. All threads watch and respect a server-wide control variable that allows the Janus main thread to stop the server and disconnect all clients.

Since the database is opened read-only by the client and thus does not need any locking, there is no performance impact on the database by using a multithreaded architecture. However, Tcl is not threadsafe and it must be ensured that under no circumstances Tcl functions are called by two threads at the same time. Therefore and for performance reasons, no Tcl functions are called in the server implementation at all but the Berkeley Socket API is used directly for network communication.[18]

The two protocol versions, two database key formats and two request types result in a total number of eight independently implemented request handlers. Six of these request handlers utilize exactly one database query to produce an answer to a request. The exceptions are the two request handlers that process array lookup requests for context+word-indexed databases. In this case, a database query must be made for every vocabulary word in order to find all n-grams having the desired linguistic context. In contrast, answering a single n-gram request with a context-indexed database requires only a single database query to be made but the n-gram must be searched in the buffer of all n-grams sharing the same linguistic context. A binary search algorithm is used to facilitate this quickly also for large buffers.

We initially started to profile the server code and implement some optimizations to buffer handling, however running a system monitor tool revealed that the CPU load on the client by far exceeds the load on the server, even if the client is running in benchmark mode which uses no Janus components except the language model. Hence, no further optimization of the server part has been performed.

[18] As an exception, debugging output is made via the Janus logging framework which uses Tcl functions internally. This debugging output is disabled by default and must not be enabled in productive use.

# Chapter 4

# Experimental Tests

## 4.1 Distribution Strategies

In this section, we want to evaluate the distribution strategies discussed in Subsection 3.2.2 on real data to discover which strategies are best suited for the application in distributed n-gram language models in matters of distribution uniformity.

### 4.1.1 Setup

We used the simulation tool presented in Subsection 3.2.3 to evaluate distribution strategies by simulating the distribution of a 61GB large 4-gram language model that contains a total of 1,840,613,688 n-grams. It has been built by interpolating several other language models using the SRILM Toolkit and uses Modified Kneser-Ney smoothing.[1]

The evaluated distribution strategies are the *sum strategy*, the *weighted sum strategy* and the *hash strategy*, as described in Section 3.2.2. For the *hash strategy*, boost::hash from the Boost C++ Library has been used. The *single word strategy* has not been implemented separately but is implicitly evaluated as a special case of the *sum strategy* and *weighted sum strategy* with $|S| = 1$. For the *weighted sum strategy*, two different values have been tested for the weighting factor $c$, namely 130,625 as the size of the vocabulary and 2,011, an arbitrarily chosen prime number, which makes for two different distribution strategies. Thus, four different base strategies have been used.

Each of these four strategies has been evaluated for 2, 3, 4, 5, 10, 25, 50, 100, 250, 500 and 1,000 servers and the most feasible word subsets $S$ for 4-grams, which are $(w_1)$, $(w_1, w_2)$, $(w_1, w_2, w_3)$, $(w_1, w_2, w_3, w_4)$, $(w_k)$, $(w_{k-1}, w_k)$, $(w_{k-2}, w_{k-1}, w_k)$, $(w_{k-1})$, $(w_{k-2}, w_{k-1})$ and $(w_{k-3}, w_{k-2}, w_{k-1})$. The application of these word selectors to shorter n-grams works as described in Subsection 3.2.3.

Since the output emitted by the simulator merely consists of the n-gram counts assigned to each server, Gini coefficients have been computed from these n-gram counts to allow for an easy comparison between distribution strategies using the same number of servers[2].

[1] Actually, the language model is the same that has been used in Subsection 4.3. More details on how the language model has been estimated can be found there.

[2] As explained previously, Gini coefficients cannot be used to compare between strategies with different server counts.

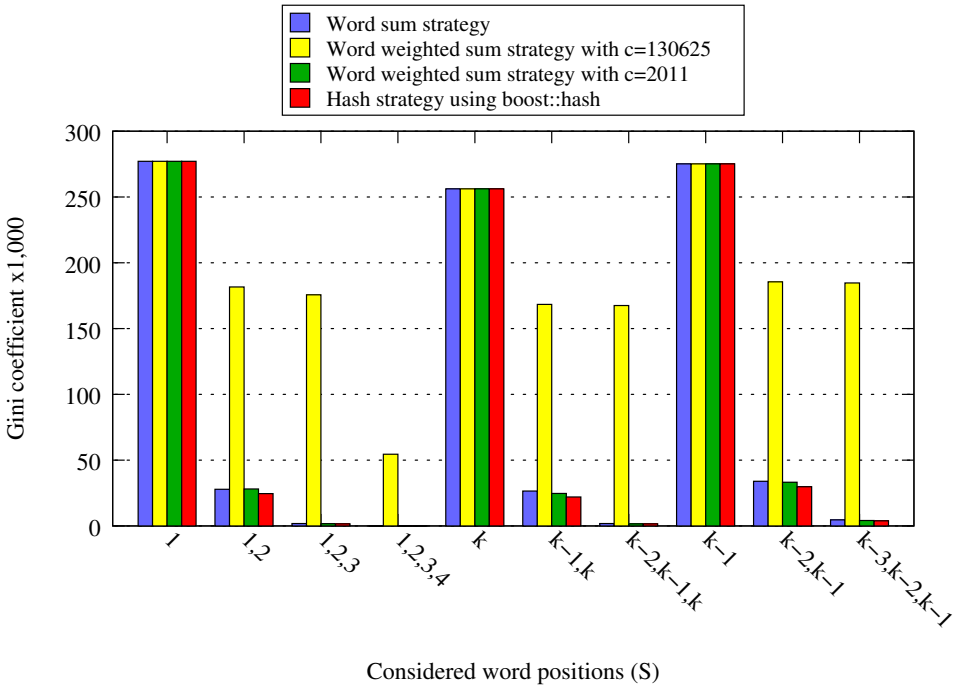


Figure 4.1: Gini coefficients for storage of n-gram language model on 100 servers

### 4.1.2 Overview

Figure 4.1 provides an overview of the computed Gini coefficients for all simulated distribution strategies. Gini coefficients in numeric form and the raw n-gram counts for the distribution strategies using two, three and five servers can be found in Appendix A.

### 4.1.3 Considered Word Positions

All four base strategies produce an identically bad distribution for $|S| = 1$ because in this case, all those strategies degrade to the same *single word strategy*. As discussed in Section 3.2.2, the *single word strategy* shows the problem that starting with a certain server number, n-grams with a single frequent vocabulary word (like "<unk>") at the observed word position will always be assigned to the same server, regardless of how many servers there are, and then dominate this server instance. This problem even increases as the number of servers grows because the number of n-grams that contain the frequent word at the observed word position constitutes a minimum space requirement for this server that cannot be further reduced as more servers are added. Thus, we will not further consider the *single word strategy* in the rest of this section.

Another interesting point is the comparison between strategies that consider the whole n-gram for distribution, i.e. $S = \{w_1, w_2, w_3, w_4\}$, and strategies that consider only the linguistic context, i.e. $S = \{w_{k-3}, w_{k-2}, w_{k-1}\}$. For the implementation of array scoring, it is favorable to exclude the current word from the distribution strategy in order to ensure that all n-grams with the same linguistic context are assigned to the same server and thus, only one server needs to be queried for such a request. Fortunately, our simulation shows that although distribution uniformity slightly decreases if the current word is excluded, it is still perfectly good enough

to not make any significant practical differences. For example, the Gini coefficient for the distribution of the 4-gram language model with the *sum strategy* using $S = \{w_{k-3}, w_{k-2}, w_{k-1}\}$ is only 0.00464.

### 4.1.4 Hash strategy

It is easily observable that the distribution generated by the *hash strategy* is usually at least as good as the distributions from the all other strategies. This is not astonishing, because the hash function used in the simulator (boost::hash from the Boost C++ Library) provides a well-established hash function that should fulfill all criteria of a good hash function, one of which is an uniform distribution of the hash values. This allows us to use the *hash strategy* as a baseline for the evaluation of the other distribution strategies like the *sum strategy* that are simpler and faster to compute.

### 4.1.5 Weighted Sum Strategy

When looking at the *weighted sum strategy*, it turns out that the choice of $c$ is crucial for the performance of the strategy. In our case, using the vocabulary size of the language model (130,625) for $c$ yields an extraordinarily bad distribution uniformity, which at least partially stems from 130,625 not being a prime number.

For example, for a total number of five servers, using a weighting factor that is a multiple of five is disastrous because when a partial sum is multiplied by five, its modulo value will always be zero afterwards. Thus, the information about the first words that are added to the weighted sum is shifted out of the range of what is considered for the distribution. To prevent this from happening, a prime number should be used for $c$. This assumption is experimentally backed by the fact that *weighted sum strategy* with $c = 2,011$ results in a much smoother distribution that is comparable to the distribution generated by the *hash strategy*.

### 4.1.6 Conclusion

Our experiment has shown that there are few practical differences between the *sum strategy*, the *weighted sum strategy* with $c = 2,011$ and the *hash strategy* using boost::hash, for distributions considering the whole n-gram or only the linguistic context. Therefore, we opted for the *sum strategy* for use in our distributed language model. Because the possibility of fast array scoring is essential for a speech decoder, we chose to consider only the linguistic context but not the current word for the strategy, i.e. $S = \{w_{k-3}, w_{k-2}, w_{k-1}\}$.

## 4.2 Timing Experiments

To estimate the influence of the various parameters of the distributed n-gram language model on overall performance of the two request types, we performed a benchmark by running timing experiments with recorded n-gram access sequences.

### 4.2.1 Setup

The benchmark routine itself, i.e. the code to rerun recorded n-gram accesses from a file, has been integrated into Janus and is controlled by a Python script to allow for easy reproduction and validation. The Python script sets the parameters, automatically performs multiple runs for every configuration while measuring the time taken and coordinates starting and stopping of client and server instances. For this experiment, network latency has been excluded by running a single server with a moderate sized language model on localhost, with the client still computing the distribution strategy. The impact of network latency will be further discussed in Subsection 4.2.4.

Rerunning an n-gram access sequence is a synthetic benchmark that selectively tests only the distributed language model and thus has the advantage of avoiding influence from other components that would make the result less clear. Additionally, the runtime of some of the other components of an ASR system is hard to predict and would increase variability in runtime[3].

All possible parameter combinations for the distributed language model have been tested in this experiment. We evaluated the two different approaches for indexing the database, as discussed in Subsection 3.3.2, the two possible access methods of Berkeley DB that are supported by our system (B+tree and extended linear hashing), and the two implemented protocol versions. In practice, the main reason to use a distributed language model are memory constrictions that prevent the language model from being loaded into the memory of a single machine. With this in mind, most of the times it makes little sense to store the lower-order n-gram portions of a language model in the distributed system, because they typically only account for a small fraction of the language model size. Since lower-order n-grams are constantly needed for backing-off, having them available locally provides a significant speedup. Therefore, we also performed benchmarks runs with bigrams and even trigrams available client-side[4].

The n-gram accesses used for the timing experiments have been collected while decoding parts of the web test set of the Quaero 2009 evaluation with a Janus instance that has been modified to write out all n-grams as they are accessed. Single n-gram lookups are tested with a sequence of 100,000 n-gram accesses[5]. For array lookups, two request sequences have been created: A larger sequence containing 5,000 accesses and a smaller one with 200 accesses, created by using the first 200 entries of the larger sequence. Creating a smaller access sequence has been necessary to allow the configurations with context+word-indexed databases that are very slow for array lookups to finish in an admissible time frame. All values shown for those configurations have already been scaled to compensate for the reduced access sequence size. Finally, it should be noted that the size of an access sequence is usually not the number of server requests, since backing-off requires two additional requests for each step.

The language model used for the timing experiments is a 4.3GB large 4-gram language model that has been built with the SRILM Toolkit and uses Modified Kneser-Ney smoothing. The text corpus used for its estimation contains 409,985,320

[3]This is largely due to unsteady I/O performance in our environment with network filesystems.

[4]As a minimum, the unigrams must be kept to establish the mapping from words to word IDs.

[5]The decoding itself does not make use of single n-gram accesses. However, the used Quaero evaluation includes a rescoring pass after decoding which generates the single n-gram requests.

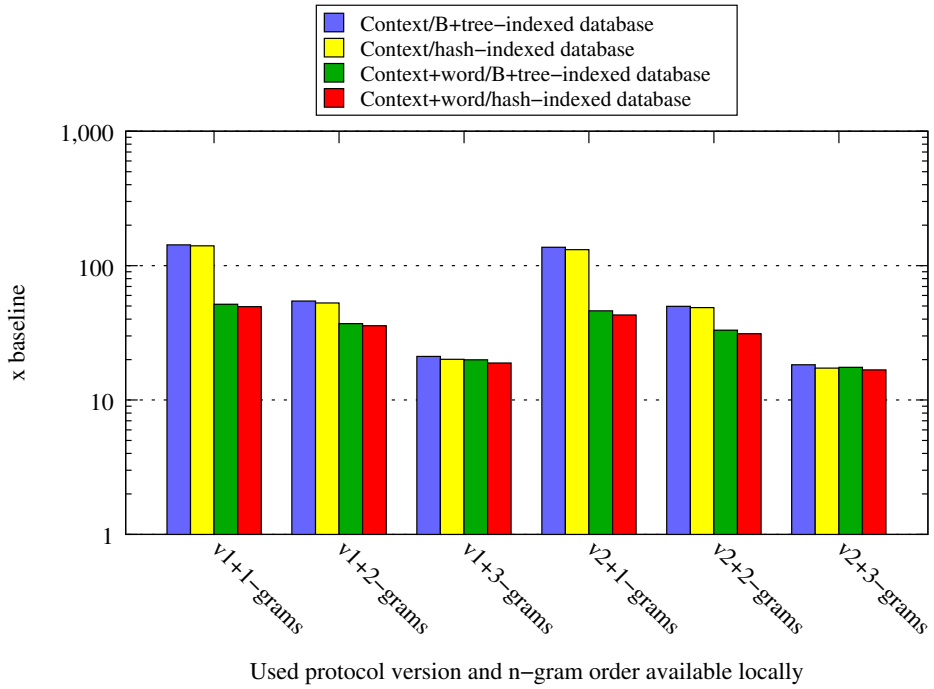


Figure 4.2: Timing results as factors of local baseline for single n-gram lookups

sentences that have been collected from the WWW by the Institute for Anthropomatics and its vocabulary consists of 129,135 words.

### 4.2.2 Overview

Figures 4.2 and 4.3 present an overview of all results from the timing experiments with the values shown as factors of the baseline. For example, a value of 1.7 means that the distributed language model was 70% slower than the baseline. The baseline is established by a benchmark run for the same n-gram access sequence with the full language model loaded locally and has been performed directly preceeding the distributed benchmark runs. All timing passes have been run five times and the values of the best run are used[6]. Numeric values can be found in Appendix B.

### 4.2.3 Discussion

From the results, it is clearly noticable that the overhead from the distributed language model declines as more n-gram orders are available locally, as we would have expected. For every n-gram order that is available to the client, two network requests are saved in case the backing-off process reaches this n-gram order.

When comparing context-indexed and context+word-indexed databases, we can see that context+word-indexed databases appear to be completely inappropriate for usage in a scenario where array lookups occur. Their bad performance for array lookups stems from the fact that with such a database, an array lookup can only be implemented by searching every possible n-gram with the desired linguistic context

[6]Using the minimum is better than using the mean (or the median) because although the cluster nodes have been allocated exclusively, there are some factors that can cause outliers, like dynamic frequency scaling. In general, the minimum is a better estimator for the actually required time for deterministic tasks than the mean.

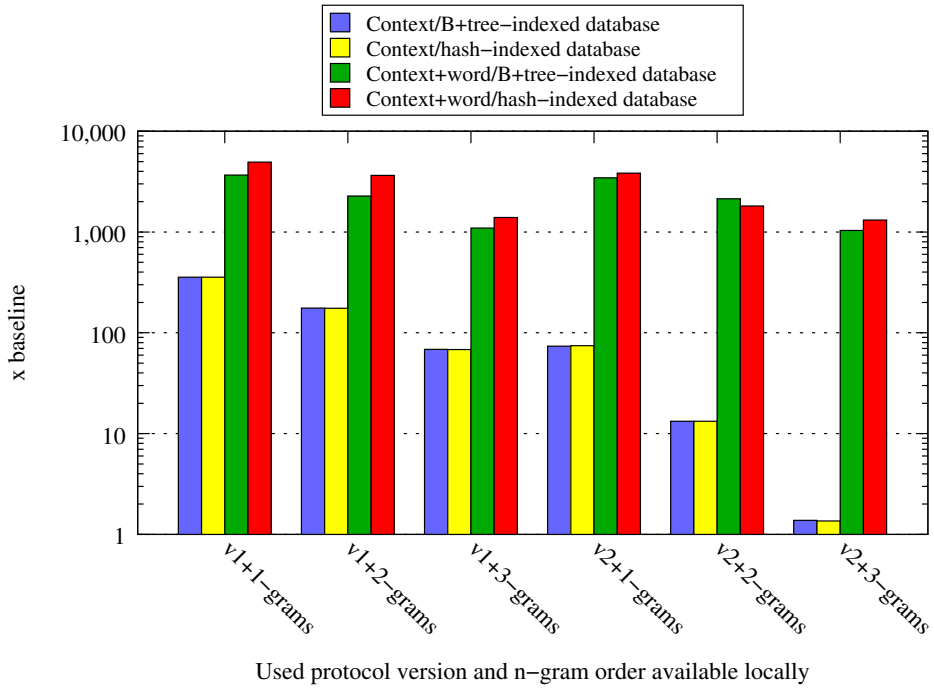


Figure 4.3: Timing results as factors of local baseline for array lookups

in the database, which makes for a number of database queries equivalent to the vocabulary size.

For single n-gram requests, context+word-indexed databases seem more suitable than context-indexed databases. The difference is not so drastic though if bigrams or even trigrams are available locally. This can be explained by the fact that very many n-grams exist for a specific linguistic context for low n-gram orders. Thus, if those n-grams orders are kept locally, a server using a context-indexed database does not incur the work to scan the large database record for such a linguistic context for the occurrence of the desired n-gram.

Comparing the two protocols, we can see that protocol v2 performs slightly better than protocol v1 for single n-gram lookups. It can be presumed that this is largely due to the missing overhead from converting numbers to text and vice versa for a binary protocol. For array lookups, protocol v2 is much more efficient than protocol v1 in terms the size of transferred data because with protocol v1, servers send a response for every vocabulary word, whether it exists as an n-gram or not. This explains the disastrous performance of protocol v1 for array lookups.

Finally, we can compare the two different access methods of Berkeley DB that can be used for the distributed language model. Although there are differences in performance, they are small and not significant for most configurations. This shows that the index accesses are not a major time factor for our databases that are kept completely in memory[7] However, in our tests, we found databases using the B+tree access method to be consistently 10%-30% smaller than databases using the hash access method. Therefore, the B+tree access method should be preferred.

[7]If the databases were read from disk, we would maybe see more drastic differences between the two access methods here, depending on the number of disk accesses that are necessary to find a database entry.

| Local 1-grams | Local 2-grams | Local 3-grams |
|---|---|---|
| 73.75 | 13.27 | 1.38 |

Table 4.1: Timing results for array lookups as factors of local baseline for a context/B+tree-indexed database when using protocol v2

### 4.2.4 Conclusion

Table 4.1 shows the results for array lookups with a practical configuration that uses a context-indexed database with the B+tree access method and protocol v2. Shown are the different values for what n-gram orders are available client-side in which the assumption that bigrams are available locally can be made even for very large language models. Storing trigrams locally as well is possible in our cluster environment for all language models used in this report, but nevertheless requires a much larger amount of memory.

What can be seen is that especially with trigrams available client-side, there is little overhead for array lookups with the distributed language model. Since the ASR decoder in Janus solely uses array lookups, performance should not decrease much when such a distributed language model is applied to speech decoding. The slowdown for single n-gram lookups is larger, but still, it should not be forgotten that the language model only accounts for a part of the computation time. Therefore, even with a slowdown of the language model by the factor 20, the application as a whole will probably be slowed down by much less than 20.

As mentioned above, network delay is not considered in this benchmark because client and server are run on the same machine to allow for automatic, scripted benchmarking which would be much harder to implement across multiple machines. For single n-gram lookups, network latency could cause an additional, maybe drastic, slowdown. But for arrow lookups as the relevant request type for decoding, this can be doubted: A local array lookup uses approximately $3.6 \cdot 10^{-4}$s = 0.36ms[8]. Network latency in our cluster has been measured by ICMP echo requests sent from one to another cluster node and round-trip time for the link was found to be 0.079ms with a standard deviation of 0.048ms[9]. Because of this, network latency should make distributed array lookups in our cluster setting slower by only around 20%.

## 4.3 Effects of Larger LMs on WER

In this section, we want to explore the effects of using very large language models on speech recognition performance, measured by the word error rate (WER) as described in Subsection 2.1.5. The task that we have used to assess the language model performance is a part of the Quaero 2009 evaluation [Stüker et al., 2010] which we have reran with a larger language model and unaltered other parameters (like preprocessing, acoustic model, etc.).

[8]The full sequence of 5,000 array lookups takes around 1.8s in the local baseline run.

[9]The measurement has been performed by running "ping -i 0.2 -c 100 i13hpc2".

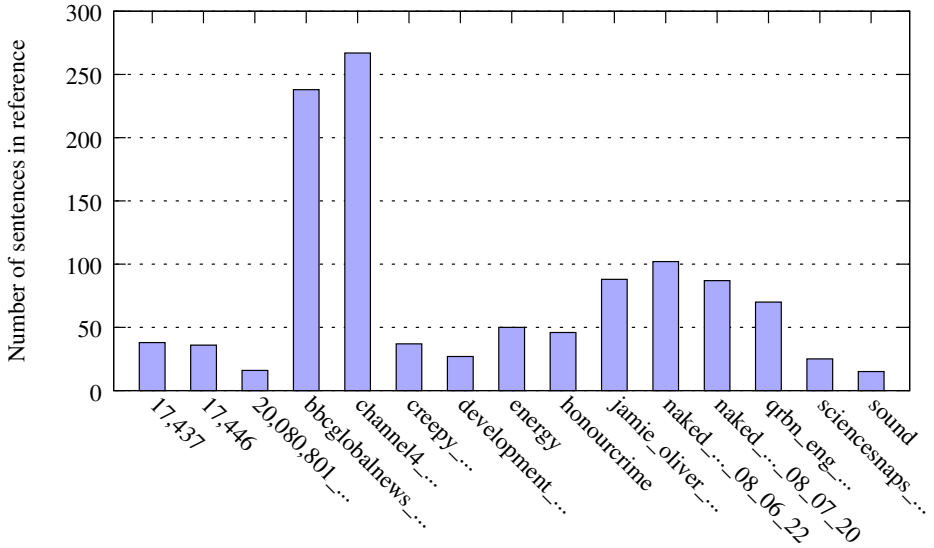


Figure 4.4: Sources used in the web test set of the Quaero 2009 evaluation

### 4.3.1 The Quaero 2009 Evaluation

Quaero is an European research and development program on multimedia classification and indexing under participation of the Karlsruhe Institute of Technology. It also includes research on the underlying technologies like multilingual automatic speech recognition in which the achieved progress of the developed systems is measured by yearly evaluations. In this experiment, we have used the evaluation of 2009 and replaced the original 3GB large language model with a 61GB large language model that will be explained later.

The Quaero 2009 evaluation is composed of two different test sets. The first test set consists of audio files gathered at different places in the World Wide Web. Figure 4.4 shows the (abbreviated) names of the sources that are contained in this test set with the sentence counts of the respective transcripts as an indicator for the influence of a source on the total word error rate. The other test set consists of speech data recorded at European Parliament Plenary Sessions (EPPS). Because such speeches typically exhibit few of the problems associated with spontaneous speech, the EPPS test set is considered to be the easier one and typically yields better word error rates.

For our experiment, only the web test set has been used. Every of its sources consists of one or more speakers which can be run independently on the computing cluster to achieve a speedup by parallelization. In contrast, utterances by the same speaker cannot be decoded independently without changing the result since speaker adaption is used.

### 4.3.2 Building a Large Language Model

The large language model for this experiment has been built by interpolating several smaller language models as described in Subsection 2.2.4. Due to limitations in the SRILM Toolkit that allows for a maximum of ten source language models, a two-step approach has been chosen, that is, first some language models are interpolated and in a second step, the result from the first interpolation is then used again for

| Language model name | Size | Weight |
|---|---|---|
| train2010.shuf.train.txt.bo4.txt | 5.6MB | 0.278557 |
| web.3-2gwl.UNK.pruned.bo4.txt | 4.4GB | 0.337259 |
| final.02.eval06.bo4p.gz.bo4.txt | 921MB | 0.137882 |
| web2-full.respell.bo4i.gz.bo4.txt | 1.1GB | 0.246301 |

Table 4.2: Language models used for first interpolation run

| Language model name | Size | Weight |
|---|---|---|
| g.4gram.bin.bo4.txt | 25GB | 0.128318 |
| q2010en.qcrap.UNK.bo4.txt.bo4.txt | 17GB | 0.0799012 |
| q2010en.odp-p1.UNK.bo4.txt.bo4.txt | 14GB | 0.00519823 |
| q2010en.odp-p0.UNK.bo4.txt.bo4.txt | 5.6GB | 0.0514753 |
| q2010en.giga.UNK.bo4.txt.bo4.txt | 9.7GB | 0.0018805 |
| web2.4-w2.4.bo4.bin.bo4.txt | 3.0GB | 0.063251 |
| smalllms.UNK.bo4.txt.bo4.txt (from first interpolation step) | 5.3GB | 0.473824 |
| kn-clean_p1_1.delchar_file.filt_line10.filt_vtab.lm.gz.bo4.txt | 17GB | 0.0959051 |
| kn-gigaword4-full-all.lm.gz.bo4.txt | 11GB | 0.0354432 |
| kn-crap-all.lm.gz.bo4.txt | 17GB | 0.0648038 |

Table 4.3: Language models used for second interpolation run

interpolation of the final language model.

In our case, four smaller language models have been interpolated in the first step and in the second step, the resulting language model has been interpolated with nine other language models. Thus, 13 language models have been used in total. Table 4.2 shows the language models used in the first step together with the weight assigned to each of them as part of the interpolation process to minimize the test set perplexity. In Table 4.3, the same type of information is presented for the second interpolation run.

Table 4.4 compares some attributes of the resulting language model with the original language model used in the baseline. It should be noted that both language models share the same vocabulary and thus have the same number of unigrams[10].

| | Baseline language model | Built language model |
|---|---|---|
| Language model order | 4 | 4 |
| Number of 1-grams | 130,625 | 130,625 |
| Number of 2-grams | 47,480,649 | 115,361,179 |
| Number of 3-grams | 59,817,610 | 523,080,964 |
| Number of 4-grams | 74,449,181 | 1,202,040,920 |
| Size in ARPABO format | 3.0GB | 61GB |
| Test set perplexity | 227.072 | 152.489 |

Table 4.4: Comparison of the built language model to the baseline language model

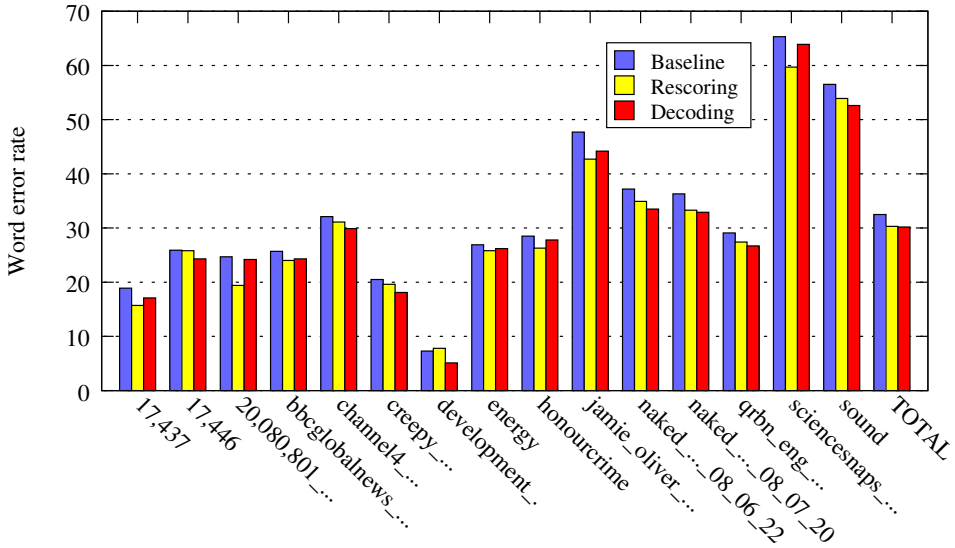


Figure 4.5: Word error rates for decoding and rescoring compared to baseline

### 4.3.3 Results

Figure 4.5 shows the results from running the evaluation by comparing the word error rate from the baseline language model to the two word error rates achieved by lattice rescoring and decoding with the larger language model. For all experiments, a rescoring pass tested different values for the language model weight (lz) and the word transition penalty (lp). The results shown here were then selected from the lz/lp matrix for the lz/lp combination with the lowest total word error rate, which was found at values of $lz = 40$ and $lp = -8$ in all three cases.

The word error rates in the figure are shown separated by test set sources and as a total value. It can be seen that usage of the larger language model systematically reduced the word error rate for almost all sources. The word error rate of the baseline of 32.5 drops by more than two points to 30.3 for the rescoring task and 30.2 for the decoding task which indicates a significant improvement in recognition accuracy. The difference between rescoring and decoding is much smaller and suggests that the lattices produced from the decoding run with the original language model already contained the hypotheses that are found during the decoding run with the larger language model in the majority of cases. Hence, the word error rate of the smaller language model is worse because other hypotheses in the lattice have been judged as more probable.

[10]If the language models used different vocabularies, it would not be permissible to compare perplexities directly.

# Chapter 5

# Conclusion

For this report, we have developed an architecture for distributed n-gram language models that allows to overcome the traditional limitations on language model size imposed by the amount of main memory available for evaluation. Because our implementation has been realized as an extension to the Janus Recognition Toolkit, applications based on this toolkit are now able to support the usage of huge distributed n-gram language models that are too big to be loaded otherwise.

In Chapter 4, we evaluated different strategies for n-gram distribution across a number of servers and found the strategy that considers the sum of the word IDs in the linguistic context of the n-gram to constitute a sufficiently smooth distribution for usage in our distributed architecture. Timing experiments of the distributed n-gram language model showed that even though our approach does bring a performance overhead with it, it is nevertheless applicable for practical usage in a speech decoder. Finally, we explored the actual benefit from using larger language models in the context of automatic speech recognition. For this purpose, we reran the Quaero 2009 evaluation using a larger, specially built language model and found a significant improvement in the word error rate for both speech decoding and lattice rescoring tasks.

## 5.1 Further Work

The work on our approach to distributed n-gram language models is not finished. Further optimizations could be made in regard to the amount of memory needed for the server databases, thus reducing the number of servers that are necessary to store a given language model. For example, it would be feasible to replace the word IDs having a fixed size of four bytes with variable-sized word IDs that are derived from the word IDs in the language model by the use of entropy encoding techniques like Huffman coding. As a result, it should be possible to lower the memory space needed to encode common words like "<s>" or "the" to one byte and spend three bytes for storing the rarely used words.

Other possible optimizations could aim at reducing the overhead of the distributed language model on speed. Many applications that use a language model can be modified to allow for some foresight of which n-gram probabilities might be needed soon. For example, a decoder for speech recognition could tentatively expand a number of promising nodes in the search graph to generate n-gram requests, and apply pruning as soon as the inquired probabilities are available. Because n-

gram requests are sent in parallel then instead of always waiting for the response to a single request, requests can be processed in parallel by the queried servers and the overhead from network latency is incurred only once. Accordingly, the speedup yielded by request batching is particularly large for applications that use a lot of single n-gram lookups but few array lookups (see Subsection 4.2.4).

Finally, with the ability to evaluate large language models, the problem of building such language models efficiently arises. For popular smoothing techniques like Modified Kneser-Ney smoothing, optimizations exist to reduce the memory demand during estimation of the language model. Nevertheless, the amount of memory needed still increases with larger corpora and language models. Therefore, the size of the text corpus that can be used is still limited with traditional tools like the SRILM Toolkit[1]. Possible solutions to this problem can either find a way to further reduce the memory consumption of the estimation process for the established smoothing techniques, or implement novel smoothing techniques that can be estimated more efficiently, while at least preserving some of the performance advantages gained by using a larger text corpus, or perform even the estimation process by itself in a distributed manner, as shown by [Brants et al., 2007].

[1]Of course, virtual memory theoretically allows to build language models of arbitrary size by swapping out memory pages to disk. In practice, however, this makes the estimation process much too slow to be advisable.

# List of Figures

# List of Tables

# Bibliography

[Brants et al., 2007] Brants, T., Popat, A. C., Xu, P., Och, F. J., Dean, J., and Inc, G. (2007). Large language models in machine translation. In *In EMNLP*, pages 858–867.

[Chen, 1998] Chen, S. F. (1998). An empirical study of smoothing techniques for language modeling. Technical report.

[Emami et al., 2007] Emami, A., Papineni, K., and Sorensen, J. (2007). Large-scale distributed language modeling. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4, pages IV–37 –IV–40.

[Goodman, 2001] Goodman, J. T. (2001). A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403 – 434.

[James, 2000] James, F. (2000). Modified kneser-ney smoothing of n-gram models. Technical report.

[Jelinek, 1991] Jelinek, F. (1991). Up from trigrams! In *Eurospeech*.

[Katz, 1987] Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, pages 400–401.

[Kneser and Ney, 1995] Kneser, R. and Ney, H. (1995). Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181 –184 vol.1.

[Olson et al., 1999] Olson, M. A., Bostic, K., and Seltzer, M. (1999). Berkeley db.

[Rosenfeld, 2000] Rosenfeld, R. (2000). Two decades of statistical language modeling: Where do we go from here. In *Proceedings of the IEEE*, page 2000.

[Seymore and Rosenfeld, 1996] Seymore, K. and Rosenfeld, R. (1996). Scalable trigram backoff language models. Technical report.

[Siivola and Pellom, 2005] Siivola, V. and Pellom, B. L. (2005). Growing an n-gram language model. In *In Proceedings of 9th European Conference on Speech Communication and Technology*, pages 1309–1312.

[Soltau et al., 2001] Soltau, H., Metze, F., Fügen, C., and Waibel, A. (2001). A one-pass decoder based on polymorphic linguistic context assignment.

[Stolcke, 2000] Stolcke, A. (2000). Entropy-based pruning of backoff language models. In *In Proceedings of the DARPA Broadcast News Transcription and Understanding Workshop*, pages 8–11.

[Stolcke, 2002] Stolcke, A. (2002). Srilm—an extensible language modeling toolkit. In *In Proceedings of the 7th International Conference on Spoken Language Processing (ICSLP 2002*, pages 901–904.

[Stüker et al., 2010] Stüker, S., Kilgour, K., and Niehues, J. (2010). Quaero speech-to-text and text translation evaluation systems. In Nagel, W. E., Kröner, D. B., and Resch, M. M., editors, *High Performance Computing in Science and Engineering '10*, pages 529–542. Springer Berlin Heidelberg.

[Zhang et al., 2006] Zhang, Y., Silja, A., and Vogel, H. S. (2006). Distributed language modeling for n-best list re-ranking. In *in Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 216–223.

# Appendix A

# Simulation Results for Distribution Strategies

Strategies in tables of this appendix chapter are abbreviated as shown in Table A.1.

| Abbrev. | Strategy |
|---|---|
| WSM | *Sum Strategy* (word sum modulo) |
| WWSM1 | *Weighted Sum Strategy* (word weighted sum modulo), $c = 130625$ |
| WWSM2 | *Weighted Sum Strategy* (word weighted sum modulo), $c = 2011$ |
| HASH | *Hash Strategy* using boost::hash from the Boost C++ Library |

Table A.1: Strategy abbreviations

The simulator described in Subsection 3.2.3 has been used to evaluate distributions of all {1,2,3,4}-grams from a 61GB large 4-gram language model to different numbers of servers using all discussed distribution strategies. The language model contains a total of 1,840,613,688 n-grams and has been interpolated from several source language models that use modified Kneser-Ney smoothing. The simulation has been run for 2, 3, 4, 5, 10, 25, 50, 100, 250, 500 and 1000 hosts.

## A.1 N-gram counts

The numbers of n-grams assigned to each server can be found in Table A.2 for 2 hosts, in Table A.3 for 3 hosts and in Table A.4 for 5 hosts.

## A.2 Gini coefficients

The Tables A.5, A.6 and A.7 show the Gini coefficients of all evaluated distribution strategies for the described 4-gram language model, multiplied by 1000. For a description of the Gini coefficient, see Subsection 3.2.3.

| Strategy | Server 1 | Server 2 |
|---|---|---|
| WSM ($w_1$) | 935429892 | 905183796 |
| WWSM1 ($w_1$) | 935429892 | 905183796 |
| WWSM2 ($w_2$) | 935429892 | 905183796 |
| HASH ($w_1$) | 905183796 | 935429892 |
| WSM ($w_1, w_2$) | 913774354 | 926839334 |
| WWSM1 ($w_1, w_2$) | 913774354 | 926839334 |
| WWSM2 ($w_1, w_2$) | 913774354 | 926839334 |
| HASH ($w_1, w_2$) | 917323016 | 923290672 |
| WSM ($w_1, w_2, w_3$) | 920204476 | 920409212 |
| WWSM1 ($w_1, w_2, w_3$) | 920204476 | 920409212 |
| WWSM2 ($w_1, w_2, w_3$) | 920204476 | 920409212 |
| HASH ($w_1, w_2, w_3$) | 920573363 | 920040325 |
| WSM ($w_1, w_2, w_3, w_4$) | 920448926 | 920164762 |
| WWSM1 ($w_1, w_2, w_3, w_4$) | 920448926 | 920164762 |
| WWSM2 ($w_1, w_2, w_3, w_4$) | 920448926 | 920164762 |
| HASH ($w_1, w_2, w_3, w_4$) | 920303398 | 920310290 |
| WSM ($w_k$) | 866770213 | 973843475 |
| WWSM1 ($w_k$) | 866770213 | 973843475 |
| WWSM2 ($w_k$) | 866770213 | 973843475 |
| HASH ($w_k$) | 973843475 | 866770213 |
| WSM ($w_{k-1}, w_k$) | 914255481 | 926358207 |
| WWSM1 ($w_{k-1}, w_k$) | 914255481 | 926358207 |
| WWSM2 ($w_{k-1}, w_k$) | 914255481 | 926358207 |
| HASH ($w_{k-1}, w_k$) | 917023616 | 923590072 |
| WSM ($w_{k-2}, w_{k-1}, w_k$) | 920437323 | 920176365 |
| WWSM1 ($w_{k-2}, w_{k-1}, w_k$) | 920437323 | 920176365 |
| WWSM2 ($w_{k-2}, w_{k-1}, w_k$) | 920437323 | 920176365 |
| HASH ($w_{k-2}, w_{k-1}, w_k$) | 920193611 | 920420077 |
| WSM ($w_{k-1}$) | 880419966 | 960193722 |
| WWSM1 ($w_{k-1}$) | 880419966 | 960193722 |
| WWSM2 ($w_{k-1}$) | 880419966 | 960193722 |
| HASH ($w_{k-1}$) | 960324347 | 880289341 |
| WSM ($w_{k-2}, w_{k-1}$) | 908919860 | 931693828 |
| WWSM1 ($w_{k-2}, w_{k-1}$) | 908919860 | 931693828 |
| WWSM2 ($w_{k-2}, w_{k-1}$) | 908919860 | 931693828 |
| HASH ($w_{k-2}, w_{k-1}$) | 916781608 | 923832080 |
| WSM ($w_{k-3}, w_{k-2}, w_{k-1}$) | 920335905 | 920277783 |
| WWSM1 ($w_{k-3}, w_{k-2}, w_{k-1}$) | 920335905 | 920277783 |
| WWSM2 ($w_{k-3}, w_{k-2}, w_{k-1}$) | 920335905 | 920277783 |
| HASH ($w_{k-3}, w_{k-2}, w_{k-1}$) | 920734273 | 919879415 |

Table A.2: N-grams counts assigned to 2 hosts from 61GB large 4-gram language model

| Strategy | Server 1 | Server 2 | Server 3 |
|---|---|---|---|
| WSM $(w_1)$ | 704267443 | 644960644 | 491385601 |
| WWSM1 $(w_1)$ | 704267443 | 644960644 | 491385601 |
| WWSM2 $(w_1)$ | 704267443 | 644960644 | 491385601 |
| HASH $(w_1)$ | 704267443 | 644960644 | 491385601 |
| WSM $(w_1, w_2)$ | 607913567 | 622102832 | 610597289 |
| WWSM1 $(w_1, w_2)$ | 619213776 | 613015252 | 608384660 |
| WWSM2 $(w_1, w_2)$ | 607913567 | 622102832 | 610597289 |
| HASH $(w_1, w_2)$ | 604781084 | 623381993 | 612450611 |
| WSM $(w_1, w_2, w_3)$ | 613640751 | 613400727 | 613572210 |
| WWSM1 $(w_1, w_2, w_3)$ | 613803793 | 613486184 | 613323711 |
| WWSM2 $(w_1, w_2, w_3)$ | 613640751 | 613400727 | 613572210 |
| HASH $(w_1, w_2, w_3)$ | 613973262 | 613374449 | 613265977 |
| WSM $(w_1, w_2, w_3, w_4)$ | 613556113 | 613549656 | 613507919 |
| WWSM1 $(w_1, w_2, w_3, w_4)$ | 613550281 | 613522183 | 613541224 |
| WWSM2 $(w_1, w_2, w_3, w_4)$ | 613556113 | 613549656 | 613507919 |
| HASH $(w_1, w_2, w_3, w_4)$ | 613535759 | 613544007 | 613533922 |
| WSM $(w_k)$ | 628333131 | 646266784 | 566013773 |
| WWSM1 $(w_k)$ | 628333131 | 646266784 | 566013773 |
| WWSM2 $(w_k)$ | 628333131 | 646266784 | 566013773 |
| HASH $(w_k)$ | 628333131 | 646266784 | 566013773 |
| WSM $(w_{k-1}, w_k)$ | 612406362 | 614654170 | 613553156 |
| WWSM1 $(w_{k-1}, w_k)$ | 618615746 | 610528590 | 611469352 |
| WWSM2 $(w_{k-1}, w_k)$ | 612406362 | 614654170 | 613553156 |
| HASH $(w_{k-1}, w_k)$ | 608733766 | 618810149 | 613069773 |
| WSM $(w_{k-2}, w_{k-1}, w_k)$ | 613916105 | 613298210 | 613399373 |
| WWSM1 $(w_{k-2}, w_{k-1}, w_k)$ | 613547712 | 613674522 | 613391454 |
| WWSM2 $(w_{k-2}, w_{k-1}, w_k)$ | 613916105 | 613298210 | 613399373 |
| HASH $(w_{k-2}, w_{k-1}, w_k)$ | 614112184 | 613178346 | 613323158 |
| WSM $(w_{k-1})$ | 647873740 | 683568268 | 509171680 |
| WWSM1 $(w_{k-1})$ | 647873740 | 683568268 | 509171680 |
| WWSM2 $(w_{k-1})$ | 647873740 | 683568268 | 509171680 |
| HASH $(w_{k-1})$ | 647873740 | 683568268 | 509171680 |
| WSM $(w_{k-2}, w_{k-1})$ | 609887685 | 618459242 | 612266761 |
| WWSM1 $(w_{k-2}, w_{k-1})$ | 621253382 | 607932909 | 611427397 |
| WWSM2 $(w_{k-2}, w_{k-1})$ | 609887685 | 618459242 | 612266761 |
| HASH $(w_{k-2}, w_{k-1})$ | 604619088 | 623714471 | 612280129 |
| WSM $(w_{k-3}, w_{k-2}, w_{k-1})$ | 613365074 | 614453910 | 612794704 |
| WWSM1 $(w_{k-3}, w_{k-2}, w_{k-1})$ | 614792297 | 614042804 | 611778587 |
| WWSM2 $(w_{k-3}, w_{k-2}, w_{k-1})$ | 613365074 | 614453910 | 612794704 |
| HASH $(w_{k-3}, w_{k-2}, w_{k-1})$ | 613766275 | 614230260 | 612617153 |

Table A.3: N-grams counts assigned to 3 hosts from 61GB large 4-gram language model

| Strategy | Server 1 | Server 2 | Server 3 | Server 4 | Server 5 |
|---|---|---|---|---|---|
| WSM ($w_1$) | 315748900 | 307390738 | 306393854 | 397503138 | 513577058 |
| WWSM1 ($w_1$) | 315748900 | 307390738 | 306393854 | 397503138 | 513577058 |
| WWSM2 ($w_1$) | 315748900 | 307390738 | 306393854 | 397503138 | 513577058 |
| HASH ($w_1$) | 307390738 | 306393854 | 397503138 | 513577058 | 315748900 |
| WSM ($w_1, w_2$) | 366443926 | 360250180 | 370250060 | 380228389 | 363441133 |
| WWSM1 ($w_1, w_2$) | 332462309 | 316671696 | 313343920 | 321144556 | 556991207 |
| WWSM2 ($w_1, w_2$) | 366443926 | 360250180 | 370250060 | 380228389 | 363441133 |
| HASH ($w_1, w_2$) | 367646908 | 366537148 | 360498195 | 371942593 | 373988844 |
| WSM ($w_1, w_2, w_3$) | 367544737 | 368456661 | 368853557 | 367863615 | 367895118 |
| WWSM1 ($w_1, w_2, w_3$) | 331462980 | 325024245 | 303866637 | 323495302 | 556764524 |
| WWSM2 ($w_1, w_2, w_3$) | 367544737 | 368456661 | 368853557 | 367863615 | 367895118 |
| HASH ($w_1, w_2, w_3$) | 368062114 | 368319870 | 368283479 | 367957794 | 367990431 |
| WSM ($w_1, w_2, w_3, w_4$) | 368112739 | 368139313 | 368130654 | 368138632 | 368092350 |
| WWSM1 ($w_1, w_2, w_3, w_4$) | 359159024 | 367086145 | 350754673 | 348129407 | 415484439 |
| WWSM2 ($w_1, w_2, w_3, w_4$) | 368112739 | 368139313 | 368130654 | 368138632 | 368092350 |
| HASH ($w_1, w_2, w_3, w_4$) | 368116108 | 368128785 | 368080297 | 368146587 | 368141911 |
| WSM ($w_k$) | 329306001 | 372894949 | 303245064 | 324129938 | 511037736 |
| WWSM1 ($w_k$) | 329306001 | 372894949 | 303245064 | 324129938 | 511037736 |
| WWSM2 ($w_k$) | 329306001 | 372894949 | 303245064 | 324129938 | 511037736 |
| HASH ($w_k$) | 372894949 | 303245064 | 324129938 | 511037736 | 329306001 |
| WSM ($w_{k-1}, w_k$) | 368971899 | 359970148 | 367422134 | 378475882 | 365773625 |
| WWSM1 ($w_{k-1}, w_k$) | 329306001 | 372894949 | 303245064 | 324129938 | 511037736 |
| WWSM2 ($w_{k-1}, w_k$) | 368971899 | 359970148 | 367422134 | 378475882 | 365773625 |
| HASH ($w_{k-1}, w_k$) | 365712958 | 366695090 | 364662904 | 369406239 | 374136497 |
| WSM ($w_{k-2}, w_{k-1}, w_k$) | 367673160 | 368166911 | 368653712 | 367798206 | 368321699 |
| WWSM1 ($w_{k-2}, w_{k-1}, w_k$) | 329306001 | 372894949 | 303245064 | 324129938 | 511037736 |
| WWSM2 ($w_{k-2}, w_{k-1}, w_k$) | 367673160 | 368166911 | 368653712 | 367798206 | 368321699 |
| HASH ($w_{k-2}, w_{k-1}, w_k$) | 367984687 | 367954226 | 368380492 | 368209091 | 368085192 |
| WSM ($w_{k-1}$) | 330702383 | 315957402 | 303454774 | 320824319 | 569674810 |
| WWSM1 ($w_{k-1}$) | 330702383 | 315957402 | 303454774 | 320824319 | 569674810 |
| WWSM2 ($w_{k-1}$) | 330702383 | 315957402 | 303454774 | 320824319 | 569674810 |
| HASH ($w_{k-1}$) | 316088027 | 303454774 | 320824319 | 569674810 | 330571758 |
| WSM ($w_{k-2}, w_{k-1}$) | 363427582 | 360179998 | 367286688 | 383572385 | 366147035 |
| WWSM1 ($w_{k-2}, w_{k-1}$) | 330702383 | 315957402 | 303454774 | 320824319 | 569674810 |
| WWSM2 ($w_{k-2}, w_{k-1}$) | 363427582 | 360179998 | 367286688 | 383572385 | 366147035 |
| HASH ($w_{k-2}, w_{k-1}$) | 365043316 | 365468004 | 360202138 | 376829768 | 373070462 |
| WSM ($w_{k-3}, w_{k-2}, w_{k-1}$) | 367324572 | 367744867 | 368636062 | 368848652 | 368059535 |
| WWSM1 ($w_{k-3}, w_{k-2}, w_{k-1}$) | 330702383 | 315957402 | 303454774 | 320824319 | 569674810 |
| WWSM2 ($w_{k-3}, w_{k-2}, w_{k-1}$) | 367324572 | 367744867 | 368636062 | 368848652 | 368059535 |
| HASH ($w_{k-3}, w_{k-2}, w_{k-1}$) | 368402332 | 368417466 | 367330798 | 368264641 | 368198451 |

Table A.4: N-grams counts assigned to 5 hosts from 61GB large 4-gram language model

| Strategy | 2 hosts | 3 hosts | 4 hosts | 5 hosts |
|---|---|---|---|---|
| WSM $(w_1)$ | 8.22 | 77.11 | 48.35 | 109.63 |
| WWSM1 $(w_1)$ | 8.22 | 77.11 | 48.35 | 109.63 |
| WWSM2 $(w_1)$ | 8.22 | 77.11 | 48.35 | 109.63 |
| HASH $(w_1)$ | 8.22 | 77.11 | 48.35 | 109.63 |
| WSM $(w_1, w_2)$ | 3.55 | 5.14 | 5.57 | 10.16 |
| WWSM1 $(w_1, w_2)$ | 3.55 | 3.92 | 5.57 | 109.33 |
| WWSM2 $(w_1, w_2)$ | 3.55 | 5.14 | 4.62 | 10.16 |
| HASH $(w_1, w_2)$ | 1.62 | 6.74 | 3.64 | 7.04 |
| WSM $(w_1, w_2, w_3)$ | 0.06 | 0.09 | 0.16 | 0.7 |
| WWSM1 $(w_1, w_2, w_3)$ | 0.06 | 0.17 | 0.16 | 111.65 |
| WWSM2 $(w_1, w_2, w_3)$ | 0.06 | 0.09 | 0.07 | 0.7 |
| HASH $(w_1, w_2, w_3)$ | 0.14 | 0.26 | 0.19 | 0.22 |
| WSM $(w_1, w_2, w_3, w_4)$ | 0.08 | 0.02 | 0.08 | 0.03 |
| WWSM1 $(w_1, w_2, w_3, w_4)$ | 0.08 | 0.01 | 0.08 | 32.82 |
| WWSM2 $(w_1, w_2, w_3, w_4)$ | 0.08 | 0.02 | 0.09 | 0.03 |
| HASH $(w_1, w_2, w_3, w_4)$ | 0.0 | 0.0 | 0.01 | 0.03 |
| WSM $(w_k)$ | 29.09 | 29.07 | 42.11 | 100.91 |
| WWSM1 $(w_k)$ | 29.09 | 29.07 | 42.11 | 100.91 |
| WWSM2 $(w_k)$ | 29.09 | 29.07 | 42.11 | 100.91 |
| HASH $(w_k)$ | 29.09 | 29.07 | 42.11 | 100.91 |
| WSM $(w_{k-1}, w_k)$ | 3.29 | 0.81 | 4.39 | 8.74 |
| WWSM1 $(w_{k-1}, w_k)$ | 3.29 | 2.93 | 4.39 | 100.91 |
| WWSM2 $(w_{k-1}, w_k)$ | 3.29 | 0.81 | 4.04 | 8.74 |
| HASH $(w_{k-1}, w_k)$ | 1.78 | 3.65 | 3.77 | 4.92 |
| WSM $(w_{k-2}, w_{k-1}, w_k)$ | 0.07 | 0.22 | 0.16 | 0.54 |
| WWSM1 $(w_{k-2}, w_{k-1}, w_k)$ | 0.07 | 0.1 | 0.16 | 100.91 |
| WWSM2 $(w_{k-2}, w_{k-1}, w_k)$ | 0.07 | 0.22 | 0.13 | 0.54 |
| HASH $(w_{k-2}, w_{k-1}, w_k)$ | 0.06 | 0.34 | 0.23 | 0.23 |
| WSM $(w_{k-1})$ | 21.67 | 63.17 | 29.24 | 118.91 |
| WWSM1 $(w_{k-1})$ | 21.67 | 63.17 | 29.24 | 118.91 |
| WWSM2 $(w_{k-1})$ | 21.67 | 63.17 | 29.24 | 118.91 |
| HASH $(w_{k-1})$ | 21.74 | 63.17 | 29.34 | 118.86 |
| WSM $(w_{k-2}, w_{k-1})$ | 6.19 | 3.1 | 7.24 | 11.01 |
| WWSM1 $(w_{k-2}, w_{k-1})$ | 6.19 | 4.82 | 7.24 | 118.91 |
| WWSM2 $(w_{k-2}, w_{k-1})$ | 6.19 | 3.1 | 6.89 | 11.01 |
| HASH $(w_{k-2}, w_{k-1})$ | 1.92 | 6.92 | 4.32 | 8.97 |
| WSM $(w_{k-3}, w_{k-2}, w_{k-1})$ | 0.02 | 0.6 | 0.04 | 0.86 |
| WWSM1 $(w_{k-3}, w_{k-2}, w_{k-1})$ | 0.02 | 1.09 | 0.04 | 118.91 |
| WWSM2 $(w_{k-3}, w_{k-2}, w_{k-1})$ | 0.02 | 0.6 | 0.1 | 0.86 |
| HASH $(w_{k-3}, w_{k-2}, w_{k-1})$ | 0.23 | 0.58 | 0.57 | 0.52 |

Table A.5: Gini coefficients (x1000) for distributions with 2, 3, 4 and 5 hosts from 61GB large 4-gram language model

| Strategy | 10 hosts | 25 hosts | 50 hosts | 100 hosts |
|---|---|---|---|---|
| WSM $(w_1)$ | 147.49 | 190.59 | 228.35 | 277.12 |
| WWSM1 $(w_1)$ | 147.49 | 190.59 | 228.35 | 277.12 |
| WWSM2 $(w_1)$ | 147.49 | 190.59 | 228.35 | 277.12 |
| HASH $(w_1)$ | 147.49 | 190.59 | 228.35 | 277.12 |
| WSM $(w_1, w_2)$ | 12.76 | 13.73 | 20.79 | 27.84 |
| WWSM1 $(w_1, w_2)$ | 110.17 | 180.54 | 181.02 | 181.63 |
| WWSM2 $(w_1, w_2)$ | 12.76 | 15.68 | 22.15 | 28.09 |
| HASH $(w_1, w_2)$ | 9.7 | 13.3 | 16.35 | 24.52 |
| WSM $(w_1, w_2, w_3)$ | 0.85 | 0.95 | 1.32 | 1.86 |
| WWSM1 $(w_1, w_2, w_3)$ | 111.67 | 175.67 | 175.69 | 175.71 |
| WWSM2 $(w_1, w_2, w_3)$ | 0.85 | 0.91 | 1.3 | 1.76 |
| HASH $(w_1, w_2, w_3)$ | 0.5 | 0.64 | 1.05 | 1.64 |
| WSM $(w_1, w_2, w_3, w_4)$ | 0.1 | 0.06 | 0.13 | 0.16 |
| WWSM1 $(w_1, w_2, w_3, w_4)$ | 32.84 | 54.44 | 54.44 | 54.45 |
| WWSM2 $(w_1, w_2, w_3, w_4)$ | 0.1 | 0.07 | 0.13 | 0.17 |
| HASH $(w_1, w_2, w_3, w_4)$ | 0.04 | 0.06 | 0.08 | 0.13 |
| WSM $(w_k)$ | 119.45 | 167.52 | 208.9 | 256.25 |
| WWSM1 $(w_k)$ | 119.45 | 167.52 | 208.9 | 256.25 |
| WWSM2 $(w_k)$ | 119.45 | 167.52 | 208.9 | 256.25 |
| HASH $(w_k)$ | 119.45 | 167.52 | 208.9 | 256.25 |
| WSM $(w_{k-1}, w_k)$ | 11.71 | 12.89 | 19.24 | 26.56 |
| WWSM1 $(w_{k-1}, w_k)$ | 101.87 | 167.52 | 167.98 | 168.4 |
| WWSM2 $(w_{k-1}, w_k)$ | 11.71 | 13.74 | 18.72 | 24.65 |
| HASH $(w_{k-1}, w_k)$ | 7.12 | 11.05 | 14.41 | 22.04 |
| WSM $(w_{k-2}, w_{k-1}, w_k)$ | 0.62 | 1.04 | 1.29 | 1.88 |
| WWSM1 $(w_{k-2}, w_{k-1}, w_k)$ | 100.95 | 167.52 | 167.54 | 167.56 |
| WWSM2 $(w_{k-2}, w_{k-1}, w_k)$ | 0.62 | 0.82 | 1.06 | 1.57 |
| HASH $(w_{k-2}, w_{k-1}, w_k)$ | 0.43 | 0.76 | 1.14 | 1.64 |
| WSM $(w_{k-1})$ | 142.62 | 184.61 | 225.01 | 275.18 |
| WWSM1 $(w_{k-1})$ | 142.62 | 184.61 | 225.01 | 275.18 |
| WWSM2 $(w_{k-1})$ | 142.62 | 184.61 | 225.01 | 275.18 |
| HASH $(w_{k-1})$ | 142.59 | 184.59 | 225.0 | 275.14 |
| WSM $(w_{k-2}, w_{k-1})$ | 15.84 | 16.81 | 25.52 | 33.92 |
| WWSM1 $(w_{k-2}, w_{k-1})$ | 120.17 | 184.61 | 185.14 | 185.5 |
| WWSM2 $(w_{k-2}, w_{k-1})$ | 15.84 | 19.36 | 26.82 | 33.27 |
| HASH $(w_{k-2}, w_{k-1})$ | 11.06 | 16.55 | 20.02 | 29.85 |
| WSM $(w_{k-3}, w_{k-2}, w_{k-1})$ | 1.46 | 1.95 | 3.05 | 4.64 |
| WWSM1 $(w_{k-3}, w_{k-2}, w_{k-1})$ | 119.01 | 184.61 | 184.66 | 184.71 |
| WWSM2 $(w_{k-3}, w_{k-2}, w_{k-1})$ | 1.46 | 1.84 | 3.23 | 4.17 |
| HASH $(w_{k-3}, w_{k-2}, w_{k-1})$ | 1.0 | 1.45 | 2.37 | 3.87 |

Table A.6: Gini coefficients (x1000) for distributions with 10, 25, 50 and 100 hosts from 61GB large 4-gram language model

| Strategy | 250 hosts | 500 hosts | 1000 hosts |
|---|---|---|---|
| WSM $(w_1)$ | 347.87 | 396.51 | 453.19 |
| WWSM1 $(w_1)$ | 347.87 | 396.51 | 453.19 |
| WWSM2 $(w_1)$ | 347.87 | 396.51 | 453.19 |
| HASH $(w_1)$ | 347.87 | 396.51 | 453.19 |
| WSM $(w_1, w_2)$ | 40.89 | 51.87 | 68.01 |
| WWSM1 $(w_1, w_2)$ | 288.98 | 290.05 | 292.61 |
| WWSM2 $(w_1, w_2)$ | 38.27 | 48.64 | 63.23 |
| HASH $(w_1, w_2)$ | 35.9 | 48.29 | 62.08 |
| WSM $(w_1, w_2, w_3)$ | 3.17 | 4.33 | 6.28 |
| WWSM1 $(w_1, w_2, w_3)$ | 284.12 | 284.13 | 284.16 |
| WWSM2 $(w_1, w_2, w_3)$ | 2.67 | 3.7 | 5.16 |
| HASH $(w_1, w_2, w_3)$ | 2.67 | 3.63 | 5.27 |
| WSM $(w_1, w_2, w_3, w_4)$ | 0.26 | 0.36 | 0.49 |
| WWSM1 $(w_1, w_2, w_3, w_4)$ | 115.62 | 115.63 | 115.63 |
| WWSM2 $(w_1, w_2, w_3, w_4)$ | 0.24 | 0.32 | 0.44 |
| HASH $(w_1, w_2, w_3, w_4)$ | 0.2 | 0.28 | 0.42 |
| WSM $(w_k)$ | 325.72 | 376.59 | 435.65 |
| WWSM1 $(w_k)$ | 325.72 | 376.59 | 435.65 |
| WWSM2 $(w_k)$ | 325.72 | 376.59 | 435.65 |
| HASH $(w_k)$ | 325.72 | 376.59 | 435.65 |
| WSM $(w_{k-1}, w_k)$ | 38.19 | 50.28 | 65.53 |
| WWSM1 $(w_{k-1}, w_k)$ | 274.43 | 275.14 | 277.44 |
| WWSM2 $(w_{k-1}, w_k)$ | 35.52 | 45.63 | 61.02 |
| HASH $(w_{k-1}, w_k)$ | 33.37 | 46.17 | 59.42 |
| WSM $(w_{k-2}, w_{k-1}, w_k)$ | 2.95 | 4.13 | 5.84 |
| WWSM1 $(w_{k-2}, w_{k-1}, w_k)$ | 274.01 | 274.02 | 274.05 |
| WWSM2 $(w_{k-2}, w_{k-1}, w_k)$ | 2.54 | 3.57 | 4.96 |
| HASH $(w_{k-2}, w_{k-1}, w_k)$ | 2.44 | 3.4 | 4.93 |
| WSM $(w_{k-1})$ | 346.45 | 395.11 | 451.1 |
| WWSM1 $(w_{k-1})$ | 346.45 | 395.11 | 451.1 |
| WWSM2 $(w_{k-1})$ | 346.45 | 395.11 | 451.1 |
| HASH $(w_{k-1})$ | 346.41 | 395.01 | 451.04 |
| WSM $(w_{k-2}, w_{k-1})$ | 49.06 | 61.6 | 76.87 |
| WWSM1 $(w_{k-2}, w_{k-1})$ | 294.23 | 294.91 | 297.09 |
| WWSM2 $(w_{k-2}, w_{k-1})$ | 45.56 | 56.34 | 71.93 |
| HASH $(w_{k-2}, w_{k-1})$ | 42.67 | 57.05 | 70.82 |
| WSM $(w_{k-3}, w_{k-2}, w_{k-1})$ | 7.33 | 10.09 | 13.74 |
| WWSM1 $(w_{k-3}, w_{k-2}, w_{k-1})$ | 293.87 | 293.91 | 294.17 |
| WWSM2 $(w_{k-3}, w_{k-2}, w_{k-1})$ | 6.43 | 8.85 | 12.39 |
| HASH $(w_{k-3}, w_{k-2}, w_{k-1})$ | 6.38 | 9.07 | 12.6 |

Table A.7: Gini coefficients (x1000) for distributions with 250, 500 and 1000 hosts from 61GB large 4-gram language model

# Appendix B

# Timing Measurements for the Distributed System

Tables B.1 and B.2 show the results from the timing experiments of the distributed language model for single n-gram and array lookups as described in Section 4.2. Each configuration is represented by a row in both tables. Five runs for every database type have been performed on the same machine, preceeded by five runs with the same request sequence and the language model loaded locally to establish a baseline. The table then shows the best performance of the five distributed runs relative to the best performance of the baseline runs. For example, a value of 1.7 means that the distributed language model was 70% slower than the local baseline.

The language model employed for the timing experiments is a 4.3GB large 4-gram language model that uses Modified Kneser-Ney smoothing and has been estimated on a text corpus gathered from the WWW by the Institute for Anthropomatics. The n-gram access sequence used for benchmarking has been recorded while decoding parts of the Quaero 2009 evaluation and consists of 100,000 accesses for single n-gram lookups and 5,000 respectively 200 accesses for array lookups[1]. The sequence of array lookups that consists of only 200 accesses has been used for the context+word-indexed databases that are very slow for array lookups, i.e. for the last two columns of Table B.2. The values in the table have already been scaled accordingly to make the results comparable.

[1]The sequence of 200 requests has been extracted from the start of the sequence containing 5,000 requests.

| | context-indexed | | context+word-indexed | |
|---|---|---|---|---|
| | B+tree index | hash index | B+tree index | hash index |
| Protocol v1, local 1-grams | 142.55 | 140.41 | 51.63 | 49.45 |
| Protocol v1, local 2-grams | 54.45 | 52.71 | 37.05 | 35.74 |
| Protocol v1, local 3-grams | 21.12 | 20.07 | 19.91 | 18.85 |
| Protocol v2, local 1-grams | 136.89 | 131.30 | 46.10 | 42.96 |
| Protocol v2, local 2-grams | 49.84 | 48.75 | 33.07 | 31.18 |
| Protocol v2, local 3-grams | 18.32 | 17.29 | 17.51 | 16.77 |

Table B.1: Timing results as factors of local baseline for single n-gram lookups

| | context-indexed | | context+word-indexed | |
|---|---|---|---|---|
| | B+tree index | hash index | B+tree index | hash index |
| Protocol v1, local 1-grams | 355.72 | 355.84 | 3668.54 | 4938.64 |
| Protocol v1, local 2-grams | 175.82 | 175.48 | 2275.81 | 3644.82 |
| Protocol v1, local 3-grams | 68.43 | 68.33 | 1095.80 | 1391.80 |
| Protocol v2, local 1-grams | 73.75 | 74.40 | 3453.57 | 3836.08 |
| Protocol v2, local 2-grams | 13.27 | 13.27 | 2140.06 | 1811.86 |
| Protocol v2, local 3-grams | 1.38 | 1.36 | 1036.45 | 1314.69 |

Table B.2: Timing results as factors of local baseline for array lookups