A Distributed System for Large-scale n-gram Language Models at Tencent

Qiang Long†, Wei Wang‡, Jinfu Deng†, Song Liu†, Wenhao Huang†, Fangying Chen†, Sifan Liu†

†Tencent, ‡National University of Singapore
†{ecalezlong, austindeng, samanthaliu, zakerhuang, aachen, stephenliu}@tencent.com
‡wangwei@comp.nus.edu.sg

ABSTRACT

n-gram language models are widely used in language processing applications, e.g., automatic speech recognition, for ranking the candidate word sequences generated from the generator model, e.g., the acoustic model. Large n-gram models typically give good ranking results; however, they require a huge amount of memory storage. While distributing the model across multiple nodes resolves the memory issue, it nonetheless incurs a great network communication overhead and introduces a different bottleneck. In this paper, we present our distributed system developed at Tencent with novel optimization techniques for reducing the network overhead, including distributed indexing, batching and caching. They reduce the network requests and accelerate the operation on each single node. We also propose a cascade fault-tolerance mechanism which adaptively switches to small n-gram models depending on the severity of the failure. Experimental study on 9 automatic speech recognition (ASR) datasets confirms that our distributed system scales to large models efficiently, effectively and robustly. We have successfully deployed it for Tencent's WeChat ASR with the peak network traffic at the scale of 100 millions of messages per minute.

PVLDB Reference Format:
Qiang Long, Wei Wang, Jinfu Deng, Song Liu, Wenhao Huang, Fangying Chen, Sifan Liu. A Distributed System for Large-scale n-gram Language Models at Tencent. *PVLDB*, 12(12): 2206 - 2217, 2019.
DOI: https://doi.org/10.14778/3352063.3352136

Keywords

n-gram Language Model, Distributed Computing, Speech Recognition, WeChat

1. INTRODUCTION

Language models estimate the probabilities of sequences of words or tokens. Typically, they assign low probabilities for rare sequences or sequences with grammar errors. For example, a language model trained over computer science articles is likely to assign a higher probability for s_1:"VLDB is a database conference" than s_2:"VLDB eases a data base conference". Language models are widely used in natural language processing applications [15], especially in applications that generate text, including automatic speech recognition, machine translation and information retrieval. Typically, language models are applied to rank the candidate outputs generated by a generator. For instance, in ASR, the generator is an acoustic model that accepts audio and outputs candidate word sequences. For candidates with similar scores from the acoustic model, e.g. s_1 and s_2, the language model is vital for selecting the correct answer.

n-gram language models are specific language models with simple and effective implementation. They estimate the probability of a sequence of words based on the statistics (e.g. frequency) of n-grams from the sequence. An n-gram is a subsequence of n words. For example, "VLDB" and "database" are 1-grams; "VLDB is" and "VLDB eases" are 2-grams. n-gram language models assign a higher probability to a sequence with frequent n-grams. The statistics are calculated against a training text corpus. The estimated probability reflects the likelihood that the sequence is generated from the training text corpus. For the sample sequence s_1 and s_2, a 3-gram model would give s_1 a higher probability because "VLDB is a" is more common than "VLDB eases a", and "a database conference" is more common than "data base conference" in computer science articles.

One big issue of using n-gram language models is its high storage cost. To produce accurate probability estimation, n-gram language models need a big n-gram set with a large n. On the one hand, with a larger n, the n-gram is more meaningful as it covers a longer context. For instance, a 1-gram model would give s_2 a larger score than s_1, because "data" and "base" are more popular than "database". In contrast, a 2-gram language model is likely to give s_1 a higher probability than s_2, since "database conference" is more common than "base conference". On the other hand, a larger n-gram set includes more n-grams (i.e. a better coverage) and thus gives better probability estimation for sequences with rare n-grams. For example, if "database conference" is not included in the n-gram set, the probability of s_1 is estimated based on its suffix, i.e., "conference". This process is called backoff (See Section 2.2) and is not accurate. Our experiment confirms that the larger model has better performance. However, it is non-trivial to store large models in the memory of a single machine and provide fast access to the statistics.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352136

Distributed computing has been applied to handle large n-gram models by distributing the statistics onto multiple nodes. We call these models distributed n-gram models. One challenge of implementing distributed n-gram models is the high communication overhead. For example, if an n-gram is not included in the text corpus, O(n) messages are incurred to estimate its probability via backoff in [5]. Considering that there could be as many as 150,000 candidate n-grams per input (sentence) [5], the communication cost becomes prohibitively expensive. Another challenge of implementing distributed n-gram models is related to the network failure, which happens quite often in a distributed system with a large amount of network communication [23]. If some n-gram messages get lost, the model would produce inaccurate estimation of the probability.

In this paper, we propose novel techniques for implementing an efficient, effective and robust distributed system to support large-scale n-gram language models. First, we cache the statistics of short n-grams, e.g. 1-grams and 2-grams, on the local node. The communication cost for low-order n-grams is thus eliminated. Second, we propose a two-level distributed index for efficient n-gram retrieval. The global level index distributes the statistics for estimating the probability of an n-gram into the same node. Consequently, only a single network message is issued for each n-gram. The local level index stores the statistics in a suffix tree, which facilitates fast search and saves storage. Third, we batch messages being sent to the same server into a single message. It significantly reduces the communication cost. In particular, each n-gram incurs less than 1 message on average. Last, we propose a cascade fault-tolerance mechanism, which uses the cached short n-grams, e.g., 2-grams, to estimate the probability of long n-grams, e.g., 4-grams, when there are minor network failures (e.g. due to packet loss) and uses a small n-gram model when there are major network failures (e.g. due to node failure).

Our contributions are summarized as follows.

1. We propose caching, indexing, and batching optimization techniques to reduce the communication overhead and speed up the probability estimation process for large-scale distributed n-gram language models.

2. We propose a cascade fault-tolerance mechanism, which is adaptive to the severity of the network failure. It switches between two local n-gram models to give effective probability estimation.

3. We implement a distributed system with the proposed techniques and conduct extensive experiments over 9 datasets using automatic speech recognition as the application. The system scales to large 5-gram models efficiently. The experimental results confirm that larger n-gram models deliver higher accuracy. We have deployed the system to support automatic speech recognition in WeChat with 100 millions of audio tracks per day, and the peak network traffic is at the scale 100 millions of messages per minute.

The rest of this paper is organized as follows: Section 2 gives the introduction about n-gram language models; Section 3 presents the details of the system including the optimization techniques and fault-tolerance mechanisms; Our experimental results are analyzed in Section 4; Section 5 reviews the related work and Section 6 concludes the paper.

2. PRELIMINARY

In this section, we first introduce how to estimate the probability of a sequence of words using n-gram language models. Then we describe the training and inference procedures.

2.1 Language Models

Given a sequence of m words[1] denoted as $w_1^m = (w_1, w_2, \cdots, w_m)$, from a vocabulary V, a language model provides the probability of this sequence, denoted as $P(w_1^m)$.

According to probability theory, the joint probability can be factorized as follows,

$$\begin{aligned} P(w_1^m) &= P(w_m|w_1^{m-1})P(w_1^{m-1}) && (1)\\ &= P(w_m|w_1^{m-1})P(w_{m-1}|w_1^{m-2})P(w_1^{m-2}) && (2)\\ &\quad ... && (3)\\ &= P(w_1)\prod_{i=2}^{m} P(w_i|w_1^{i-1}) && (4)\end{aligned}$$

Typically, in the natural language processing (NLP) applications, the probability in Equation 4 is combined with the score from the sequence generator to rank the candidate sequences. For example, ASR systems use acoustic models to generate the candidate sentences. The acoustic score is combined with the score (Equation 4) from the language model to rank the candidate sentences. Those with grammar errors or strange word sequences will get smaller scores from the language model and thus be ranked at the lower positions.

An n-gram language model assumes that a word in a sequence only depends on the previous n-1 words. Formally,

$$P(w_i|w_1^{i-1}) = P(w_i|w_k^{i-1}) \quad (5)$$

where $i \in [1, m], k = \max(1, i - n + 1), P(w_1|w_1^0) = P(w_1)$, and $w_k^{i-1} = (w_k, w_{k+1}, \cdots, w_{i-1})$.

Applying the assumption (Equation 5, called Markov assumption [15]) in Equation 4, we have

$$P(w_1^m) = P(w_1)\prod_{i=2}^{m} P(w_i|w_k^{i-1}) \quad (6)$$

For example, a 3-gram model estimates the probability of a sequence of 4 words as,

$$\begin{aligned} &P(w_1, w_2, w_3, w_4)\\ &= P(w_4|w_2, w_3)P(w_3|w_1, w_2)P(w_2|w_1)P(w_1)\end{aligned}$$

Once we have the conditional probabilities of n-grams (where $n = 1, 2, \cdots$), we can apply Equation 6 to get the joint probability. Given an n-gram w_1^n, the conditional probability $P(w_n|w_1^{n-1})$ represents the likelihood that w_1^{n-1} is followed by w_n in the training text corpus. Therefore, the conditional probability is estimated based on the frequency of w_1^n and w_1^{n-1}. Equation 7 gives the formal definition, where $C()$ is the frequency count. For example, if (w_2, w_3) is usually followed by w_4, then $C(w_2, w_3, w_4)$ is close to $C(w_2, w_3)$ and $P(w_4|w_2, w_3)$ is close to 1.

[1]We assign each word an integer ID (i.e. the index of the word in the vocabulary) and manipulate words using their IDs instead of the word strings.

$$P(w_n|w_1^{n-1}) = \frac{C(w_1^n)}{C(w_1^{n-1})} \quad (7)$$

The probability for a 1-gram, e.g. $P(w_1)$ in Equation 6, is estimated as follows

$$P(w_1) = \frac{C(w_1)}{\sum_{w \in V} C(w)}, \forall w_1 \in V$$

2.2 Smoothing Techniques

One problem of the frequency based estimation (Equation 7) is that when an n-gram does not appear in the training corpus, e.g. when the training corpus is small, the frequency count is 0, i.e. $C(w_1^n) = 0$. Consequently, the conditional probability (Equation 7) is zero, which results in 0 for the joint probability (Equation 6). Smoothing is a technique for resolving this "zero frequency" problem. There are two popular types of smoothing methods, namely backoff models and interpolated models. The general idea is to move some probability mass from frequent n-grams to rare (or unseen) n-grams and estimate the probabilities of them based on the **suffix grams**. A suffix gram of an n-gram w_1^n is $w_i^n, 1 \leq i \leq n$.

Backoff smoothing model:

$$P(w_n|w_1^{n-1}) = \begin{cases} \alpha_{w_1^n}, & \text{if } C(w_1^n) > 0 \\ \gamma_{w_1^{n-1}} P(w_n|w_2^{n-1}), & \text{otherwise} \end{cases} \quad (8)$$

In Equation 8, $\alpha_{w_1^n}$ represents the discounted probability of (frequent) n-grams. For instance, a popular smoothing technique, called Kneser-Ney smoothing, computes $\alpha_{w_1^n}$ as

$$\alpha_{w_1^n} = \frac{\max(C(w_1^n) - D, 0)}{C(w_1^{n-1})}$$

, where D is another hyper-parameter for deducting the probability mass. If the n-gram is a rare n-gram, the probability of the suffix gram w_2^n is exploited, i.e., $P(w_n|w_2^{n-1})$, which itself may depend on $P(w_n|w_3^{n-1})$. The backoff process continues until there is a suffix gram w_i^n with $C(w_i^n) \geq k$ ($i \leq n$); k is a hyper-parameter, e.g., k=1; if such suffix gram exist, it is called the **hit gram** of the backoff process; otherwise, 0 is returned. For rare low-order grams, e.g., 1-gram w_n, their probabilities are increased using the deducted probability mass from frequent high-order grams. The coefficient $\gamma_{w_1^{n-1}}$, called the backoff weight, is trained to make the probabilities form a proper distribution, i.e., $\sum_{w \in V} P(w|w_1^{n-1}) = 1$. Closed-form solutions are available [6]. In addition, if $C(w_1^{n-1}) = 0$, $\gamma_{w_1^{n-1}}$ is set to 1.

Interpolated Kneser-Ney smoothing:

$$P(w_n|w_1^{n-1}) = \frac{\max(C(w_1^n) - D, 0)}{C(w_1^{n-1})} + \gamma_{w_1^{n-1}} P(w_n|w_2^{n-1}) \quad (9)$$

In our experiments, we apply the interpolated version of Kneser-Ney smoothing, which is defined in Equation 9. With simple manipulation (Section 2.8 of [6]), the interpolated models (e.g. Equation 9) can be converted into the same formulation as Equation 8. Therefore, the two smoothing methods share the same inference algorithm[2]. In the rest of this paper, we use the backoff model (Equation 8) to introduce the inference algorithm.

[2]Note that the training algorithms are different.

2.3 Training and Inference

The training procedure of n-gram language models counts the frequencies over a training text corpus to estimate the conditional probabilities (Equation 4) of all 1-grams, 2-grams, $\cdots$, n-grams appearing in the training corpus and compute the coefficients. For example, a 5-gram model has the statistics of 1-grams to 5-grams. Note that only grams appearing in the training corpus are processed by the training algorithm. There are two reasons: a) for a given n and a word vocabulary V, the complete n-gram set whose size is $|V|^n$ would consume a huge amount of memory when V is large; b) Some grams are never used, e.g. "is be do". There is no need to pre-compute and store their statistics. In this paper, we focus on the performance of the inference stage and thus skip the details about the training. Note that the training over a large text corpus (e.g. of Terabytes) is also very challenging. Like [5], we use distributed framework (i.e. Spark) to accelerate the training process.

The inference procedure accepts the n-gram w_1^n generated by other modules e.g., the acoustic model of ASR systems, as the input, and returns $P(w_n|w_1^{n-1})$. If the n-gram appears in the training corpus, its conditional probability has already been computed during training, which can be retrieved directly; otherwise, we estimate the probability online using the smoothing technique (Equation 8).

All probabilities and coefficients generated from the training stage is saved on disk and is load into memory during inference. ARPA [27] is a common file-format for n-gram language models. One n-gram takes one line of the ARPA file, which consists of three fields, namely $\log_{10} P(w_n|w_1^{n-1})$, the n-gram string w_1^n, and $\log_{10} \gamma(w_1^n)$. For example, the line *"-0.3 ASR is -0.1"* means $log_{10}P(is|ASR) = -0.3$ and $log_{10}\gamma(ASR\ is) = -0.1$. For ease of presentation, in the rest of this paper, we assume the backoff weight and the probability (instead of the log version) are stored in the ARPA file. The probability stored in the ARPA file is denoted as $\bar{P}(w_n|w_1^{n-1})$ to be different from the one in Equation 5. For any n-gram w_1^n, if it appears in the ARPA file, then $P(w_n|w_1^{n-1})$ is just $\bar{P}(w_n|w_1^{n-1})$, i.e., Equation 10. Otherwise, it is calculated recursively according to Equation 8 until there is a hit gram that appears in the ARPA file. The result is one of the equations between Equation 10 and 14.

$$\bar{P}(w_n|w_1^{n-1}) \quad (10)$$
$$\gamma(w_1^{n-1}) \quad \bar{P}(w_n|w_2^{n-1}) \quad (11)$$
$$\gamma(w_1^{n-1})\gamma(w_2^{n-1}) \quad \bar{P}(w_n|w_3^{n-1}) \quad (12)$$
$$\cdots$$
$$\gamma(w_1^{n-1})\gamma(w_2^{n-1})\cdots\gamma(w_{n-2}^{n-1}) \quad \bar{P}(w_n|w_{n-1}) \quad (13)$$
$$\gamma(w_1^{n-1})\gamma(w_2^{n-1})\cdots\gamma(w_{n-2}^{n-1})\gamma(w_{n-1}) \quad \bar{P}(w_n) \quad (14)$$

Given an n-gram w_1^n, Algorithm 1 implements a recursive function to compute $P(w_n|w_1^{n-1})$ following Equation 8. $\mathcal{A}$ denotes all the n-grams from the ARPA file. If w_i^n is included in the ARPA file, the algorithm simply retrieves the entry for it and returns the probability (Line 2); otherwise, it computes the probability via backoff in Line 4 ($\gamma_{w_i^{n-1}} = 1$ if $w_i^{n-1} \notin \mathcal{A}$). For 1-grams not included in the ARPA file, 0 is returned (Line 6). Note that it is expensive to retrieve

Algorithm 1 CalcProb(w_i^n)

1: **if** $w_i^n \in \mathcal{A}$ **then**
2: **return** $P(w_n|w_i^{n-1})$
3: **else if** $i < n$ **then**
4: **return** $\gamma(w_i^{n-1}) \times$ CalcProb(w_{i+1}^n)
5: **else**
6: **return** 0
7: **end if**

the gram w_i^n (Line 1-2) when $\mathcal{A}$ is large. In Section 3.2.2, we propose a special index to accelerate the search.

2.4 Problem Definition

In this paper, we assume that the n-gram language model has been trained offline. *Our system loads the ARPA file(s) into memory and conducts online inference to compute the probabilities of sequences of words generated by other modules*, e.g., the acoustic models for ASR. We present the techniques we proposed to do the inference efficiently, effectively and robustly for large-scale n-gram language models.

3. DISTRIBUTED SYSTEM

Typically, larger n-gram language models are more accurate in probability estimation. However, when the language model has too many (long) n-grams, e.g. a large ARPA file with 400 GB storage, we cannot load the whole file into the main memory of a single node. If we only load part of the ARPA file into main memory and put the rest (n-grams) on disk, the inference process is very expensive as it has to retrieve statistics of n-grams from the disk.

We adopt distributed computing to resolve the issue by partitioning the ARPA file onto multiple nodes, where each node's memory is large enough to store its shard. One such node is called a **server**. Correspondingly, there is another type of node called **client**. The client node runs the following tasks. First, it generates sequences of words using other modules, e.g. the acoustic model of ASR; Second, it sends request messages to servers to retrieve the conditional probability of each n-gram from the sequences; An n-gram is a query. if the query is not stored on the servers, the serves apply backoff to estimate the probability as illustrated in Algorithm 1. Last, the client combines the probabilities according to Equation 6. This process is called **decoding**, which is elaborated in Section 3.3.

In this section, we firstly introduce our optimization techniques to reduce the communication overhead, including caching, distributed indexing and batch processing. After that, we describe our cascade fault-tolerance mechanism against communication failures. Our system is denoted as DLM, short for distributed language model.

3.1 Caching

Caching is widely used in databases for system optimization. In DLM, we cache the statistics of short n-grams. In particular, 1-grams and 2-grams are cached on the client node. Caching long n-grams, e.g., 3-grams, would further reduce the network cost. However, it also incurs more storage cost. Section 4.2.2 compares the storage cost and the communication reduction in details. By caching these statistics, we not only reduce the communication cost but also improve the other components of the system. Specifically, we fulfill three objectives as follows,

1. Reducing the number of network messages. For 2-gram and 1-gram queries, the conditional probabilities are estimated using the local cached statistics. Hence, there is no need to send messages to remote servers.
2. Supporting hashing over 2-grams in Algorithm 2 (Section 3.2.1), which makes the data on servers more balanced.
3. Supporting fault-tolerance, which will be discussed in Section 3.4.

3.2 Distributed Index

The distributed index in DLM consists of two levels, namely, the global index on the client node and the local index on the server nodes.

3.2.1 Global Index

With the local cache on the client node, the client estimates the probability for 1-gram and 2-gram queries locally. For a long n-gram query, the client uses the global index to locate the servers that store the statistics for computing $P(w_n|w_1^{n-1})$ (according to Algorithm 1), and then sends messages to these severs. If we can put all required statistics on the same server, only a single message is sent to the server. The global index is built (Algorithm 2) to achieve this goal. There are two arguments, namely the n-gram set $\mathcal{A}$ from the ARPA file and the number of servers B. For each 1-gram and 2-gram, we replicate it on every node (Line 3) for backoff estimation. For other n-grams ($n \geq 3$), we distribute the conditional probability based on the hash of $w_{n-2}w_{n-1}$ (Line 5-6). The backoff weight $\gamma(w_1^n)$ is distributed based on $w_{n-1}w_n$ (Line 7-8). The same hash function is shared in Line 5 and Line 7.

Algorithm 2 BuildGlobalIndex($\mathcal{A}, B$)

1: **for** each n-gram $w_1^n \in \mathcal{A}$ **do**
2: **if** $n \leq 2$ **then**
3: put $< w_1, \bar{P}(w_1), \gamma(w_1) >$ or $< w_1w_2, \bar{P}(w_2|w_1), \gamma(w_1^2) >$ on every server and client
4: **else**
5: s1 = hash($w_{n-2}w_{n-1}$) $\%B + 1$
6: put $< w_1^n, \bar{P}(w_n|w_1^{n-1}) >$ on server s1
7: s2 = hash($w_{n-1}w_n$)$\%B + 1$
8: put $< w_1^n, \gamma(w_1^n) >$ on server s2
9: **end if**
10: **end for**

During inference, if $n \leq 2$, we do the estimation on the local client without sending any messages; otherwise, we we simply send the request message for $P(w_n|w_1^{n-1})$ to the server with ID $hash(w_{n-2}w_{n-1})\%B+1$. Algorithm 1 is run to get the probability using one of the equations between Equation 10 and 14. The global index guarantees that for any n-gram w_1^n ($n > 2$), all statistics used in Algorithm 1 can be accessed from the same server. For instance, we denote $\bar{w}$ ="a database". According to Algorithm 2, the backoff weights of all grams (e.g., "is a database") ending with $\bar{w}$ (Line 7 and 8) are put on the same server as the conditional probabilities of all grams (e.g., "is a database conference") whose third and second last word is $\bar{w}$ (Line 5 and

6). In addition, $\bar{P}(w_n|w_{n-1})$ and $\gamma(w_{n-2}^{n-1})$ (resp. $\bar{P}(w_n)$ and $\gamma(w_{n-1})$) used by Equation 13 (resp. Equation 14) are replicated on ever server (Line 3). To conclude, all statistics used in Equation 10 and 14 are available from the same server. Consequently, the communication cost of Algorithm 1 is reduced to only one single network message per n-gram.

Load-balancing Algorithm 1 also works if we only cache 1-grams. However, caching both 1-grams and 2-grams results in better load-balance. We explain it as follows. Considering that some words are very popular, like "is" and "the", distributing n-grams based on the hash of a single word is likely to result in imbalanced load among server nodes. For example, the server storing n-grams ending with 'the' would have a larger shard and receive more messages from the clients. Since the distribution of 2-grams is more balanced than that of 1-grams (i.e. words), distributing n-grams based on two words ($w_{n-1}w_n$ or $w_{n-2}w_{n-1}$) in Algorithm 1 would result in more balanced workload, which is verified in the experiment section. In fact, we can hash against 3-grams to make the distribution even more balanced. However, by hashing against 3-grams, we need to cache 3-grams on the local client node and replicate 3-grams on every server node, which increases the storage cost significantly.

3.2.2 *Local Index*

Once the server node receives the message for an n-gram query from the client, it searches against the local index to get the statistics and combines them to compute $P(w_n|w_1^{n-1})$. A local index is built on every server for efficient retrieval of the statistics. We use suffix tree as the index structure, where each edge represents one or more words from the vocabulary and each node represents a sequence of words by concatenating the edges. For instance, the node for n-gram w_1^n is denoted as *node* w_1^n. For simplicity of presentation, we use this notation w_1^n for both the word sequence and the node in the suffix tree. The backoff weights and conditional probabilities are associated with the nodes as shown in Figure 1.

Algorithm 3 BuildLocalIndex($\mathcal{G}, \mathcal{P}$)

```
1: root = CreateNode()
2: U = {}                                   ▷ dictionary for 1-grams
3: for each < w_1^n, P̄(w_n|w_1^{n-1}) > ∈ P do
4:     if n > 1 then
5:         node=TraverseOrInsert(root, (w_{n-1}, ..., w_1))
6:         node.insert(< w_n, P̄(w_n|w_1^{n-1}) >)
7:     else                                 ▷ Uni-grams
8:         U ←< w_n, P̄(w_n) >
9:     end if
10: end for
11: for each < w_1^n, γ(w_1^n) > ∈ G do
12:     node=TraverseOrInsert((w_n, w_{n-1}, ··· , w_1))
13:     node.g=γ(w_1^n)
14: end for
15: return root, U
```

We build the local index following Algorithm 3. Each server has two sets of statistics generated from Algorithm 2, namely the backoff weights denoted as $\mathcal{G} = \{< w_1^n, \gamma(w_1^n) >\}$ and the probabilities denoted as $\mathcal{P} = \{< w_1^n, \bar{P}(w_n|w_1^{n-1} >\}$. For each pair from $\mathcal{P}$, if it is for a 1-gram, we simply store it in a dictionary U (Line 8); otherwise, we traverse the tree following the path $w_{n-1}, w_{n-2}, \cdots, w_1$. If we reach the leaf node before finishing the path, a new edge and node for the rest words are created. The last visited node is returned (Line 5). The conditional probability is inserted into a sorted array with w_n as the key (Line 6), which enables binary search.

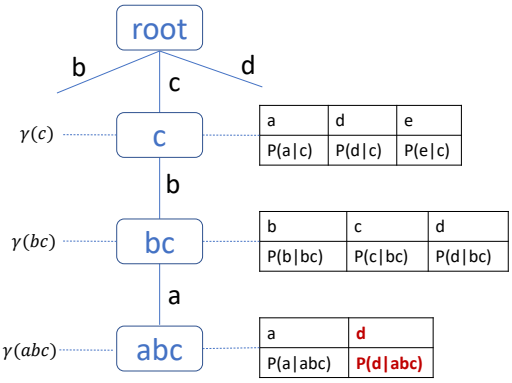


Figure 1: Insert a 4-gram "abcd" into the local index.

For each pair from $\mathcal{G}$, we traverse the tree following the path $w_n, w_{n-1}, \cdots, w_1$ (Line 7), which is the reverse of the full n-gram. New nodes are inserted during the traversal. $\gamma(w_1^n)$ is assigned to the returned node (Line 8). Note that each node may have multiple associated probabilities; however, it can only have one backoff weight. This is because the probabilities of all n-grams sharing the same prefix (i.e. w_1^{n-1}) are inserted into the same node; whereas the backoff weight is associated with the node corresponding to the full n-gram, which is unique. Figure 1 gives one example of inserting a 4-gram into the index. The conditional probability of this 4-gram is at the right bottom (bold text).

During inference, we run Algorithm 4 to estimate the conditional probability. It implements Algorithm 1 against the suffix tree [3]. Algorithm 4 traverses along (w_{n-1}, $\cdots$, w_2, w_1) (Line 2). It stops when there is no edge for the next word or when w_1 is reached. The returned *path* is a stack of visited nodes. The root node is at the bottom and the last visited node is at the top. Then it pops the nodes out one bye one. All types of n-grams are handled by computing the probability using one of the equations between Equation 10 and 14. The detailed analysis of Line 3-14 is explained in the Appendix A.

3.3 Batch Processing

Typically, the generator generates multiple candidate words for each output position. Consequently, there are many candidate sentences. When the language model is applied to rank the candidate words immediately after the candidates of one position are generated, it is called on-the-fly rescoring; when it is applied to rank the candidate sentences after finishing all positions, it is called multi-pass rescoring. Compared with multi-pass recoring, on-the-fly rescoring incurs smaller storage cost and is more efficient as it maintains a small set of candidate sentences by filtering words with low scores. In DLM, we adopt on-the-fly rescoring.

[3]Algorithm 1 ignores the data structure for storing and accessing the statistics.

Algorithm 4 SearchLocalIndex($w_1^n, root$)

```
1: g = 1, p = 0
2: path = Traverse(root, (w_{n-1}, w_{n-2}, ..., w_1))
3: while path.top() != root do
4:     node = path.pop()
5:     if w_n ∈ node then                          ▷ w_n found
6:         p =BinarySearch(node, w_n)
7:         break
8:     else
9:         g* = node.g
10:    end if
11: end while
12: if p == 0 then p = U[w_n]        ▷ U is from Algorithm 3
13: end if
14: return g * p
```

For each new position, the generated words are concatenated to each candidate sentence pair-wisely to construct new candidate sentences[4]. To estimate the joint probability of each new candidate sentence, we need to get the conditional probability of the last n-gram (the probabilities of other n-grams are already computed). At the beginning, the candidate sentences are short, with only a single word "START" representing the start of a sentence. The first set of query n-grams are thus 2-grams like "START I". As the sentences become longer, the query n-grams become longer, e.g., 5-grams. If we maintain at most K candidate sentences and generate W words for each position, we then need to query KW n-grams, i.e., KW messages. This number could be very large, like 10,000, which results in many network messages. To reduce the communication cost, we propose to batch the messages sent to the same server into a single message.

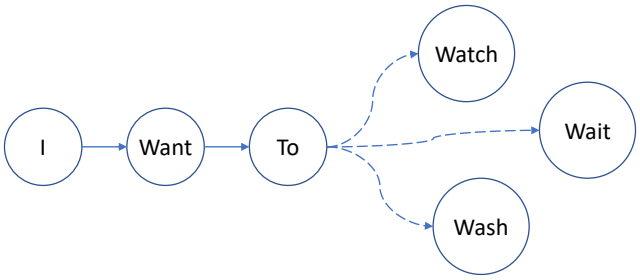


Figure 2: An example of n-grams sharing the same prefix.

If two n-grams share the same prefix, according to Algorithm 4 (and Equation 10-14), it is likely that they share similar access patterns over the local index for the statistics to compute the conditional probabilities. Suppose we use a 4-gram model to rank the candidate words in Figure 2, i.e, "Watch", "Wait" and "Wash". Even with the distributed index, we need one message for each of the 3 4-grams. Since they share the same prefix "I Want To", the three messages are sent to the same server according to Line 5 of Algorithm 2. To reduce the communication cost, we merge the three messages into a single message. Once the message is received by the server node, Algorithm 4 traverses along the reverse of the common prefix "I Want To", and then traverses back to get the statistics of the 3 n-grams respectively. The results are put into a single message and sent back to the client. For this example, we reduce the number of messages from 1 per n-gram to 1/3 per n-gram. In other words, we can save 2/3 cost.

[4]They are actually the prefixes of the sentences.

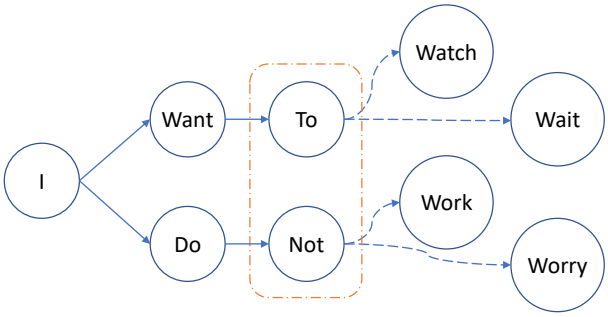


Figure 3: An example of n-grams distributed to the same server.

For n-grams not sharing the same prefix, we are still able to merge their request messages as long as they are sent to the same server. Suppose "Want To" and "Do Not" have the same hash value, then in Figure 3, according to Line 5 of Algorithm 1, the messages for all 4-grams can be merged (batched) into a single message. On the server side, the n-grams sharing the same prefix are processed together by Algorithm 4. The two batching methods are orthogonal and thus can be combined together.

3.4 Fault-tolerance

Fault-tolerance is vital for distributed applications. Depending on the severity of the network failures, we provide a cascade fault-tolerance mechanism. Figure 4 shows the architecture of our system. The connection tester keeps sending heartbeats to servers. when the failure rate is high, e.g., more than 0.1% messages time out or the network delay doubles, we redirect all messages to a local **small language model**. In our experiment, we use a small (5-gram) model with 50 GB statistics, which is pruned from the complete model via entropy-based pruning [25]. When the failure rate is low, we send the messages through the normal route. Particularly, if the message reaches the server successfully, Algorithm 4 is executed to estimate the probability; if the message times out (e.g. due to network failure), we estimate the probability locally using the cached 2-grams (backoff via Equation 8) on the client. We call the cached n-grams a **partial n-gram model**. Note that the partial model and the small model are trained separately. Hence, we cannot mix their statistics to estimate the sentence probability (Equation 6).

4. EXPERIMENTAL STUDY

In this section, we evaluate the proposed optimization techniques for ASR over multiple datasets. The results show that our distributed language model, denoted as DLM, scales efficiently to large models (e.g. with 400 GB statistics) with small communication overhead. Specifically, the overhead of DLM increases only 0.1 second when the input audio increases one second, which is small enough for real-time deployment in WeChat.

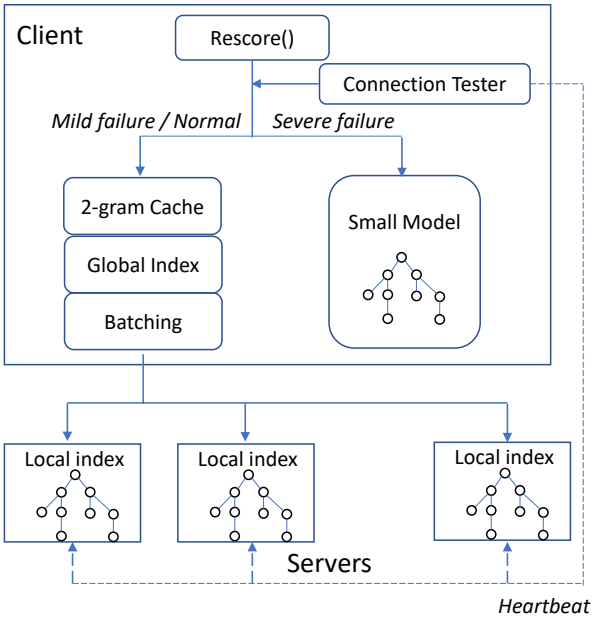


Figure 4: System architecture of DLM.

Table 1: Size and recording environment of the test datasets.

Dataset	Size (hours)	Recording Environment
test-1	15	spontaneous speech
test-2	15	mix
test-3	17	spontaneous speech
test-4	2	spontaneous speech
test-5	10	mix
test-6	4	spontaneous speech
test-7	40	mix
test-8	20	noise
test-9	29	noise

4.1 Experimental Setup

4.1.1 Hardware

We conduct the experiments over a cluster with a Intel(R) Xeon(R) CPU E5-2670 V3 (2.30GHz) and 128 Giga Bytes (GB) memory on each node. The nodes are connected via 10 Gbps network cards. We use the open source message passing library *phxrpc*[5].

4.1.2 Datasets

We collect a large text corpus (3.2 Terabytes) to train a 5-gram language model using interpolated Kneser-Ney smoothing. After training, we get a 500 GB ARPA file, which is further pruned to generate a 50GB, 100GB, 200GB and 400GB model for experiments. Without explicit description, the experiments are conducted over the 200GB model, which is the currently deployed model. To evaluate the performance of the language models, we use ASR as our application. 9 audio datasets are collected as the test data. We name these 9 datasets as test-1 to test-9, with details shown in Table 1. We can see that the test datases are recorded under different environments and vary in terms of the length. *mix* indicates that the audio has both reading and spontaneous speech. *noise* means that the audio was recorded in public area with noise or the speaker was far away from the recording device. In conclusion, our test data has a good coverage of real-life application scenarios.

[5] https://github.com/Tencent/phxrpc

4.1.3 Evaluation metrics

We evaluate the effectiveness, efficiency and storage cost of DLM in supporting ASR. For effectiveness, we use the **word error rate (WER)**, which is a common metric of ASR. To calculate the WER, we manually generate the reference word sequence (i.e. the ground truth) for each audio track from the test datasets. Given an input audio track, we align the the word sequence generated by the model with the the reference word sequence, and then apply Equation 15 to compute WER. In Equation 15, D, S and I are the numbers of deletion, substitution and insertion operations respectively involved in aligning the two sequences following Levenshtein distance. N is the total number of words in the reference sequence. For example, to convert the prediction word sequence in Figure 5 to the reference sequence, we need $D = 1$ deletion operation, $S = 2$ substitution operations, $I = 1$ insertion operation. The corresponding WER is $\frac{1+2+1}{5} = 0.8$.

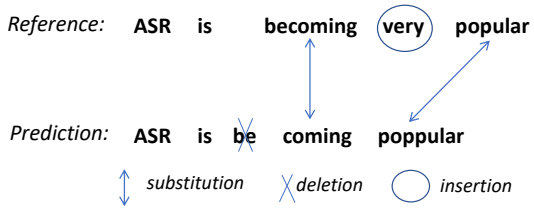


Figure 5: An example of aligning the prediction word sequence with the reference sequence.

$$WER = \frac{D + S + I}{N} \tag{15}$$

For efficiency, we measure the average processing time spent by the language model per second input audio. Distributed n-gram model incurs communication overhead. We also measure the number of network messages, which are the major cause of the communication overhead.

4.2 Scalability and Efficiency Evaluation

We firstly study the overall efficiency and scalability of DLM. After that, we do a breakdown analysis of the performance of each optimization technique.

4.2.1 Scalability test

We test the scalability of DLM in terms of throughput, i.e., the number of messages processed by the servers per second. We use a large model with 400GB data, and measure the number of messages per minute handled by DLM. By controlling the rate of messages from clients to servers, the maximum CPU utilization rate on all servers is kept at 75%. We run two sets of workloads: a) real workload, denoted as DLM-Real, whose messages are generated by the decoder without any moderation; b) balanced workload, denoted as DLM-Balanced, whose messages are manually balanced among all servers of DLM. For DLM-Real, the distribution

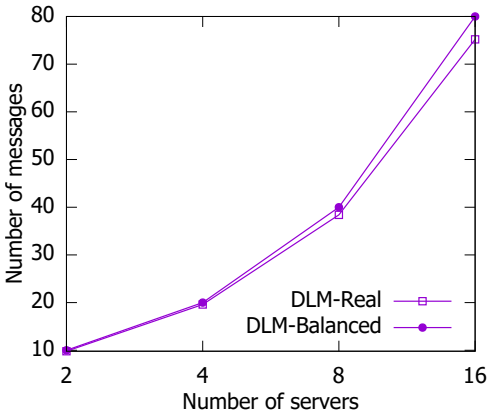
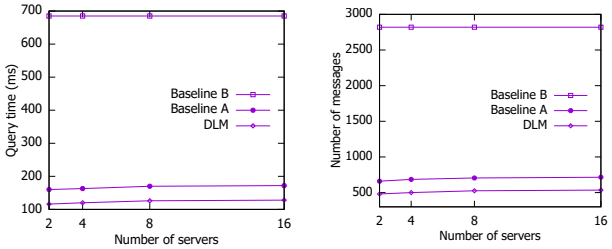


Figure 6: Scalability test by measuring the throughput.

of messages to servers are imbalanced. Hence, some servers would receive fewer messages and the overall throughput is affected. The result in Figure 6 shows that DLM-Balanced and DLM-Real scale well with only a small difference. The good scalability is due to the proposed optimization techniques, which will be analyzed in details in the following subsections.

A scalable system should minimize the overhead incurred when there are more servers. In Figure 7, we measure the overhead by the change of querying time and number of messages generated per second audio when the number of servers is increased. We compare DLM with two baseline approaches, denoted as **Baseline A**[5] and **Baseline B**[20]. Baseline A distributes both the backoff weights and the probabilities based on the hash of the last two words of each n-gram. Batch processing is briefly mentioned in the paper with the details omitted. We use the same batch processing method of DLM for Baseline A. Baseline B distributes the probabilities and backoff weights based on the hash of all words in the n-gram; in addition, it caches short grams locally. We implement Baseline B following the same cache mechanism as DLM, i.e., cache 1-grams and 2-grams.



(a) (b)

Figure 7: Comparison with baselines.

The processing time per one second input audio of DLM and the baselines is shown in Figure 7a. We can see that when the number of servers increases, DLM and Baseline A scale well with little overhead (or change) in terms of querying time. The querying time of Baseline B remains high and almost the same for different number of servers. This can be explained by the number of messages from the client to servers per second audio (Figure 7b). Note that the y-axis

Table 2: Time breakdown per communication.

Stage	Time (μs)
Decoding & encoding	60
Transmission	130
Local index search	10

of Figure 7b is different to that in Figure 6, which stands for the messages[6] processed by the servers per second. Algorithm A and DLM incur fewer messages than Algorithm B mainly because of batch processing. For Algorithm A and DLM, when the number of servers increases, the chance for two messages sent to the same server decreases. Therefore, the number of messages and the processing time increases. In addition, DLM's distributed index guarantees that all statistics of one n-gram is on a single server; However, for Baseline A, it can only guarantee that all probabilities used in the backoff process of an n-gram are distributed to the same server (because the suffix grams in Equation 8 share the same last two words), whereas the backoff weights may be distributed to different servers. Consequently, fewer messages are generated by DLM than Algorithm A.

To understand where the time goes, We measure the time spent for each stage of a single network communication in Table 2. We can see that, with the optimized local index, most time is spent on message transmission, encoding and decoding, which requires the optimization of the message passing library. In this paper, we focus on reducing the number of messages. Next, we study the effectiveness of our proposed optimization techniques in communication reduction.

4.2.2 *Caching*

DLM caches the 2-grams and 1-grams on the client side. Consequently, 1-gram and 2-gram queries are answered locally without sending messages to server nodes. From Table 3, we can see that 16.09% 2-gram queries can be answered directly. Therefore, we reduce 16.09% messages (without considering batch processing) by caching 2-gram queries locally. There is only one 1-gram query, i.e., "START", which has no influence on the performance.

Table 3: Distribution of queries.

n-gram	2	3	4	5	total
percent (%)	16.09	13.51	13.49	56.89	1
with hit gram (%)	16.09	10.6	4.2	2.5	33.39

We do not cache 3-grams and 4-grams because they are very large as shown in Table 4. Moreover, only a small portion of 3-gram and 4-gram queries can be answered directly without backoff. The rest 3-gram and 4-gram queries have to be estimated by the remote servers. In other words, caching 3-grams and 4-grams brings small reduction in communication cost. The size of 5-grams is not large, because the training algorithm prunes many 5-grams that can be estimated using 4-grams or 3-grams accurately in order to save the storage. We do not cache 5-grams because only 2.5% 5-grams queries (Table 3) can be answered using the 5-grams

[6]The messages could be sent from multiple clients to keep the servers' CPU utilization rate at 75%.

Table 4: Distribution of n-grams in the ARPA file.

n-gram	1	2	3	4	5
number (million)	0.13	600	3600	2000	400
size (GB)	≈ 0	18	108	66	15

in the ARPA files; other 5-gram queries need backoff, which incurs network communication.

4.2.3 Distributed Index

We conduct an ablation study to evaluate the performance of the global and local index of DLM (Section 3.2) separately. We do not consider the batching and caching optimization for this experiment. The global index of DLM puts all data for estimating the probability of an n-gram in the same server node and thus reduces the number of messages to 1 per n-gram. To evaluate the effect of this reduction, we list the query distribution in Table 3. There is only one 1-gram query, i.e., "START". Hence, we do not list it in the table. We can see that 33.39% of all queries have hit grams. In other words, these 33.39% queries do not need backoff and incur only one message per query. The rest queries need at least one time backoff, which incur at least one more message. With the global index, we always issue one message per query, which save at least (1-33.39%)=66.61% cost.

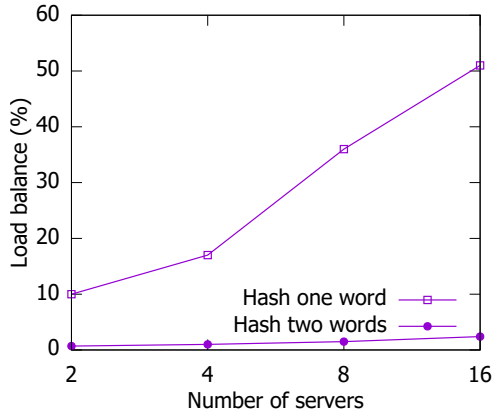


Figure 8: Load-balance comparison

The global index partitions the data in the ARPA file onto multiple servers. We evaluate the load-balance among the servers. The load-balance is calculated as

$$\frac{max(S) - avg(S)}{avg(S)}$$

, where S denotes the set of local index sizes on all server nodes, $max(avg)$ computes the largest (resp. average) index size. Figure 8 compares the load balance of the global index by hashing over a single word and two words. We can see that the distribution of data among servers are more balanced when the hash is done over two words.

To evaluate the performance of the local index of DLM, i.e., the suffix tree, we create an n-gram query set and compare the search time using the our local index versus using a baseline index which stores the conditional probability and the backoff weight using two separate hash functions (with the whole n-gram as the key). Table 5 shows that DLM's

Table 5: Evaluation of the local index.

	DLM local index	Hash Index
Number of searches	732	1663
Time per search (μs)	6.5	4.0
Total time (ms)	4.76	6.65

Table 6: Storage cost comparison.

	ARPA	DLM	WFST
Size (GB)	200	100	200

local index is much faster than the hash index (see the Total time). This is mainly because that the local index saves the time by reducing the number of searches. To be specific, on average, each n-gram needs 2.27 times of backoff to reach the hit gram. DLM searches down the local index (suffix tree) once per n-gram and then traverses back to get all the information required by the backoff process. In contrast, the baseline method calls the hash index repeatedly during the backoff process (Algorithm 1) to get the backoff weights and probabilities. Although the speed of the hash index is fast per search, the overall efficiency gap is dominated by the big difference in the number of searches. Note that the time per search in Table 5 is smaller than that in Table 2. This is because in Table 2, we measure the time for per message, which includes a batch of n-grams. Some of these n-grams may share the same prefix (see Section 3.3) and are thus processed together. Consequently, the search time is longer. The time in Table 5 is only for a single n-gram.

The local index also saves storage because the suffix tree stores the prefix words only once among all n-grams sharing this prefix. Figure 6 compares the storage cost of ARPA file, DLM (i.e., the local index) and another popular data structure for storing the language model, i.e. WFST[22]. We can see that DLM saves nearly half of the space.

4.2.4 Batch processing

Batch processing proposed in Section 3.3 merges the messages sent to the same server into a single message to reduce the overhead. Figure 9 compares DLM with batch processing and DLM without batch processing over two sets of n-gram queries. The y-axis is the total number of messages for each query set. We can see that batch processing reduces the number of network messages significantly. In fact, al-

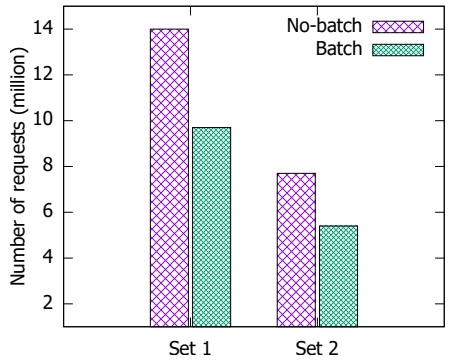


Figure 9: Evaluation of batch processing.

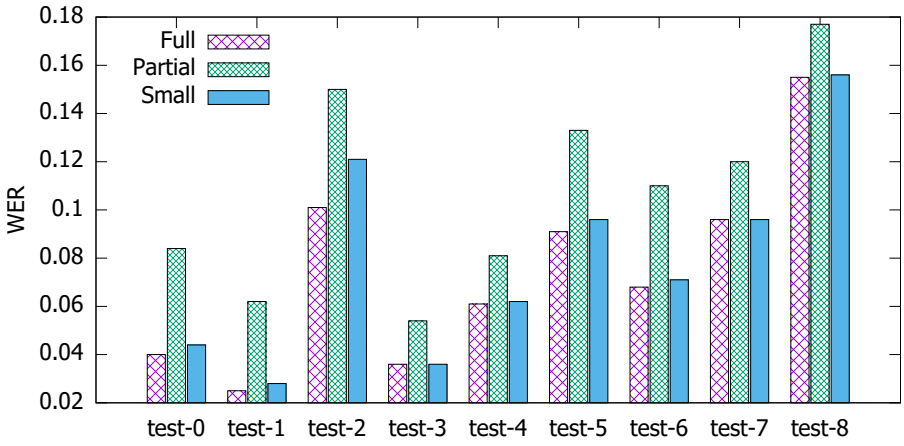


Figure 10: Performance of different models for fault-tolerance.

most half of all messages are eliminated. Consequently, it saves much time. The big gap between DLM and Baseline B in Figure 7 is also mainly due to batch processing.

4.3 Fault-tolerance Evaluation

Our distributed system has two back-up models for fault-tolerance, which help to make the system robust against network failures. However, since we are using a partial or a small language model, the recognition performance is likely to degrade. To compare the accuracy of these models, we run each model over the 9 test datasets independently. Figure 10 shows the WER of the small, partial and full model. The full model's ARPA file has 200 GB data. The partial model includes all 1-grams and 2-grams of the full model. The small model is a 5-gram language model trained over a subset of the text corpus. The ARPA file size is 50 GB.

We can see that the full model has the smallest WER, which confirms our assumption that the larger model has better performance. In addition, the small model is better than the partial model and is competitive with the full model on some test datasets. This is because the small model is pruned from the full model via sophisticated pruning strategy[25], whereas the partial model is simply pruned by removing 3-grams, 4-grams and 5-grams. Note that the partial model is only invoked for mild network failures, which has limited influence on the overall WER at runtime. Although the small model has similar performance as the full model on multiple test datasets (environments), it is still worth to use the full model as the full model is more robust across different environments, including noisy environment with multiple speakers, which is very important for commercial deployment.

5. RELATED WORK

5.1 Language Models

Many research papers [6] on n-gram models focus on the smoothing techniques, which resolve the zero-probability problem for n-grams not seen in the (small) training corpus. Among them, Kneser-Ney smoothed 5-gram models [17] provide a strong baseline.

Big web data[7] has been collected for training large n-gram models. Big data brings significant improvement on accuracy [5]; however, it introduces challenges of storing large models in the memory of a single machine. To handle large n-gram models, distributed computing is applied for both training [5, 1] and inference [28, 5]. For example, Brants et al. [5] use Map-Reduce to train an n-gram language model over a terabyte corpus. Each mapper processes one chunk of the corpus to count the statistics, and the reducers aggregate the statistics to generate the ARPA file(s).

In this paper, we focus on the online inference stage of n-gram models and assume that models have already been trained. Zhang et al.[28] and Emami et al.[8] partition the training corpus onto multiple server nodes. Each server trains an n-gram language model separately. During inference, the client sends messages to all servers to fetch the statistics and merge them to estimate the n-gram probability. Brants et al.[5] partition the original ARPA file by hashing the last two words of each n-gram. Using a new smoothing technique, called stupid backoff, they can reduce the number of network messages to 1 per n-gram; however, for normal smoothing techniques like Kneser-Ney, O(n) messages have to be sent per n-gram. Mandery [20] studies different partitioning approaches and uses the sum of the IDs of the prefix (n-1) words as the hash key. This strategy results in good load-balance. However, it does not guarantee that the probabilities and backoff weights of an n-gram query are on the same server. In the worst case, a single n-gram query can incur $2n-1$ messages. Our distributed index works for most popular smoothing methods and reduces the network messages to 1 per n-gram.

Neural language models (NLM)[4, 24] that use neural networks to estimate the probability for LM become popular. NLM, especially the recurrent neural network based LMs[21], have the potential to use longer context for probability estimation than n-gram LMs, whose context length is at most n. However, the large vocabulary size and big training corpus bring computation challenge for NLM [14]. Large scale LMs [14, 26, 12] show that NLMs and n-gram LMs complement each other and the performance degrades when they are used in isolation.

[7] `https://ai.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html`

5.2 Automatic Speech Recognition

In recent years, we have witnessed the big changes of ASR. Hidden Markov model (HMM) + Gaussian mixture model (GMM) were used as the acoustic model for ASR for a long time[15]. Around 2010s, deep neural network (DNN) replaced GMM, and DNN+HMM became the state-of-the-art[11]. Since 2013, researchers[10, 9, 19, 2] have been trying to train end-to-end recurrent neural network models for ASR by removing the HMM. For both the traditional models and the end-to-end models [19, 2], n-gram LMs are integrated to help rank the candidate words and sentences.

5.3 Optimization for Machine Learning

Despite the fast improvement of computation hardware, optimization from the algorithm and system perspective is still demanding as the models are becoming more complex and the datasets are increasing at the same time. Database researchers have been working on exploiting the optimization techniques from database systems like batch processing, indexing, caching, etc. for optimizing machine learning systems or tasks[18, 13, 7, 3, 16]. Clipper[7] proposes a general system for machine learning inference, which batches the messages received from users and feed them into the model for higher throughput; MacroBase [3] accelerates the analytics of fast data streams by applying the machine learning models only over sampled data. NoScope [16] provides specific optimization for video search. It trains a sequence of cheap video classifiers tailored for the query and exploits a cost-optimizer to search for the most efficient video classifier with the given accuracy constraint. DLM shares some ideas with these approaches, however, which are developed for other applications instead of n-gram language modelling.

6. CONCLUSION

n-gram language models are ubiquitous in NLP applications. Distributed computing enables us to run larger n-gram language models with better accuracy. However, it is challenging to make the system scalable due to the communication cost and network failures. Towards the challenges, we present a distributed system to support large-scale n-gram language models. To reduce the communication overhead, we propose three optimization techniques. First, we cache low-order n-grams on the client node to serve some requests locally. Second, we propose a distributed index to process the rest requests. The index distributes the statistics required by each n-gram query onto the same server node, which reduces the number of messages per n-gram to only one. In addition, it builds a suffix tree on every server for fast retrieval and estimation of the probabilities. Third, we batch all messages being sent to the same server node into a single message. Besides the efficiency optimization, we also propose a cascade fault-tolerance mechanism, which switches to local small language models adaptively. The experiment results confirm that our system scales to large n-gram models efficiently, effectively and robustly.

7. ACKNOWLEDGEMENT

The work in this paper is supported by the Tencent-NUS Collaborative Research Grant, the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity RD Programme (No. NRF2016NCR-NCR002-020), and Singapore Ministry of Education Academic Research Fund Tier 1.

8. REFERENCES

[1] C. Allauzen, M. Riley, and B. Roark. Distributed representation and estimation of wfst-based n-gram models. In *Proceedings of the ACL Workshop on Statistical NLP and Weighted Automata (StatFSM)*, pages 32–41, 2016.

[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR*, abs/1512.02595, 2015.

[3] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Prioritizing attention in fast data. In *SIGMOD*, pages 541–556, 2017.

[4] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, Mar. 2003.

[5] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *EMNLP-CoNLL*, pages 858–867, Prague, Czech Republic, June 2007.

[6] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394, 1999.

[7] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, Boston, MA, 2017. USENIX Association.

[8] A. Emami, K. Papineni, and J. Sorensen. Large-scale distributed language modeling. In *ICASSP*, volume 4, pages IV–37–IV–40, April 2007.

[9] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *ICML - Volume 32*, ICML'14, pages II–1764–II–1772. JMLR.org, 2014.

[10] A. Graves, A. Mohamed, and G. E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[11] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.

[12] S. Ji, S. V. N. Vishwanathan, N. Satish, M. J. Anderson, and P. Dubey. Blackout: Speeding up recurrent neural network language models with very large vocabularies. *CoRR*, abs/1511.06909, 2015.

[13] J. Jiang, F. Fu, T. Yang, and B. Cui. Sketchml: Accelerating distributed machine learning with data sketches. In *SIGMOD*, SIGMOD '18, pages 1269–1284, New York, NY, USA, 2018. ACM.

[14] R. Józefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[15] D. Jurafsky and J. H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[16] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. *PVLDB*, 10(11):1586–1597, 2017.

[17] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *ICASSP*, volume 1, pages 181–184 vol.1, May 1995.

[18] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *SIGMOD*, SIGMOD '18, pages 1493–1508, New York, NY, USA, 2018. ACM.

[19] A. L. Maas, A. Y. Hannun, D. Jurafsky, and A. Y. Ng. First-pass large vocabulary continuous speech recognition using bi-directional recurrent dnns. *CoRR*, abs/1408.2873, 2014.

[20] C. Mandery. Distributed n-gram language models : Application of large models to automatic speech recognition. 2011.

[21] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur. Extensions of recurrent neural network language model. pages 5528 – 5531, 06 2011.

[22] M. Mohri, F. Pereira, and M. Riley. Weighted finite-state transducers in speech recognition. *Comput. Speech Lang.*, 16(1):69–88, Jan. 2002.

[23] Y. Shen, G. Chen, H. V. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor. Fast failure recovery in distributed graph processing systems. *PVLDB*, 8(4):437–448, 2014.

[24] D. Shi. A study on neural network language modeling. *CoRR*, abs/1708.07252, 2017.

[25] A. Stolcke. Entropy-based pruning of backoff language models. *CoRR*, cs.CL/0006025, 2000.

[26] W. Williams, N. Prasad, D. Mrva, T. Ash, and T. Robinson. Scaling recurrent neural network language models. In *ICASSP*, pages 5391–5395, April 2015.

[27] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland. *The HTK book*. 01 2002.

[28] Y. Zhang, A. S. Hildebrand, and S. Vogel. Distributed language modeling for n-best list re-ranking. In *EMNLP*, EMNLP '06, pages 216–223, Stroudsburg, PA, USA, 2006.

APPENDIX

A. ANALYSIS OF ALGORITHM 4

Denote each node using the words from that node to the root. For example, if *Traverse* completes all words, then the last node is node w_1^{n-1}. The execution of Line 3 to Line 13 has four states,

1. If the top of the stack is node w_1^{n-1}, and $< w_n, \bar{P}(w_n|\ w_1^{n-1}) >$ is associated with this node, the probability is retrieved at Line 6 and returned at Line 14 ($g = 1$). This case corresponds to Equation 10.

2. If the top of the stack is node w_k^{n-1}, where $1 \leq k < n$, and $< w_n, \bar{P}(w_n|\ w_k^{n-1}) >$ is not associated with this node, it indicates that $\forall i < k$ node w_i^{n-1} does not exist, and thus $w_i^{n-1} \notin \mathcal{A}$. According to Section 2.2, $\gamma(w_i^{n-1}) = 1$. Algorithm 4 accumulates the backoff weight at Line 9 ($node.g = 1$) and checks the next node in the stack, i.e. node w_{k+1}^{n-1}. The execution goes to (2), (3) or (4).

3. If the top of the stack is node w_l^{n-1}, and $< w_n, \bar{P}(w_n|\ w_l^{n-1}) > (1 < l < n)$ is associated with this node, the probability is retrieved and the result is returned at Line 14. If the execution is from state (2), we have

$$\begin{aligned} g * p &= \prod_{i=k}^{l-1} \gamma(w_i^{n-1}) \bar{P}(w_n|w_l^{n-1}) \\ &= \prod_{i=1}^{l-1} \gamma(w_i^{n-1}) \bar{P}(w_n|w_l^{n-1}) = P(w_n|w_1^{n-1}) \end{aligned}$$

   The second equation is derived as $\gamma(w_i^{n-1}) = 1, \forall i < k$. The last equation is according to the l-th equation between Equation 10 and Equation 14. If the execution is directly from the initial state, we have

$$\begin{aligned} g * p &= 1 * \bar{P}(w_n|w_l^{n-1}) \\ &= \prod_{i=1}^{l-1} \gamma(w_i^{n-1}) \bar{P}(w_n|w_l^{n-1}) = P(w_n|w_1^{n-1}) \end{aligned}$$

   The second equation is derived as $\forall i < l$ node w_i^{n-1} does not exist, which indicates $w_i^{n-1} \notin \mathcal{A}$ and $\gamma(w_i^{n-1}) = 1$.

4. If the top node is the root, Line 12 is then executed. Similar to state (3), we can prove the returned value is just the conditional probability. If the execution is from (2), the returned value is

$$\begin{aligned} g * p &= \prod_{i=k}^{n-1} \gamma(w_i^{n-1}) \bar{P}(w_n) \\ &= \prod_{i=1}^{n-1} \gamma(w_i^{n-1}) \bar{P}(w_n) = P(w_n|w_1^{n-1}) \end{aligned}$$

   The last equation is according to Equation 14. If the execution is from the initial state, the returned value is

$$\begin{aligned} g * p &= 1 * \bar{P}(w_n) \\ &= \prod_{i=1}^{n-1} \gamma(w_i^{n-1}) \bar{P}(w_n) = P(w_n|w_1^{n-1}) \end{aligned}$$

From the analysis, we can see Algorithm 4 handles all types of n-grams by computing the probability using one of the equations between Equation 10 and 14.