Name : Siwei Chen

Uni : sc4574

Haskell-Words-Auto-Completion

Problem Definition

The auto-completion feature is aimed to help users type what they want more easily. Users can type a fraction of words or sentences. Then the program will predict what should follow next. There are serval kinds of auto-completion for different tasks. Here we use terminologies in ElasticSearch to describe them.

Completion Suggester

Given a string p and int k, find most possible k words beginning with p in the corpus.

Example:

Input: p = "pre", k = 3

Output: t = ["prefix", "preview", "prelude"]

Phrase Suggester

Given a string p and int k, find most possible k words following after p in the corpus.

Example:

Input: p = "some", k = 3

Output: t = ["apples", "bananas", "hints"]

Other constraints

- All the words should only consist of lowercase characters (a-z), hyphen (-), apostrophe (').
- If the number of valid candidates is less than k, return all the candidates.

Serial Implements

Serial Implement for Completion Suggester

An intuitive criterion of possibility is the appearance frequency of words. We can easily get a straightforward solution.

1. Count word frequency in the corpus using the MapReduce-like algorithm in HW3.
2. Filter words that starting with the string p.
3. Build a min-heap with size k to save k most frequent words.

If we use a list to store the result of step 1, this Linear running time algorithm will become noticeably slow when the corpus grows large enough. Even if step 1 is executed only once. Iterating all the words is still unaffordable for a single query. So we can use the prefix tree to store the words and their frequency. This will yield a logarithm solution.

Serial Implement for Phrase Suggester

The scale of this problem will be around the square of the previous one. So the improvement of the algorithm should be more noticeable.

1. Count frequency f of word pairs (u, v).
2. Build a weighted directed graph to store the results in the form of the adjacency list.
3. Trim the vertex with a degree more than k by the weight of edges to reduce complexity.
4. Access the node and its neighborhood in constant time using the adjacency list.

Most runtime in this part will be spent on the first 3 steps. After that, step 4 will be constant time for each query.

Parallelism to Accelerate Processing

Parallelism for Word Frequency Counter

In both Suggesters, counting word frequency is the heaviest duty part. But since MapReduce is designed for distributed systems. We can easily divide the corpus by the prefix characters and distribute the workload of calculation to different threads.

Parallelism for Completion Suggester

For every single query, after positioning the subtree with the given prefix string p, by sharding the subtree using the character, the traversal of this subtree can be parallel and each time a thread takes a child node to search top k frequent words in it. Then aggregate and filter the result of all the child nodes.

Parallelism for Phrase Suggester

Since the query of Phrase Suggester can be seen as a constant-time operation. Most of the improvement will be derived from the parallelism of the Word Frequency Counter, which is also the parallelism of MapReduce.

Extra Part: Auxiliary Functions

Functions in this part will make this project more like a productive program. Due to less relevance to the theme of parallelism, these features have lower priority.

Saving/Loading Model in Databases

Once the analysis of the corpus done, the model can be huge. Keep the model in memory is not economic and the scale of the corpus is limited to memory size. Thus saving the model in databases and retrieve them partially on demand will an efficient way.

Interactive Interface

An interactive interface will make this project more completed and easier to demonstrate how the program makes sense. This post would be helpful to achieve the goal neatly with Monad.