Distributed Corpus Search

Radoslav Rábara, Pavel Rychlý, Ondřej Herman
Masaryk University, Faculty of Informatics
Botanická 68a, 602 00 Brno
{xrabara,pary,xherman1}@fi.muni.cz

Abstract

Available amount of linguistic data raises fast and so do the processing requIrements. The current trend is towards parallel and distributed systems, but corpus management systems have been slow to follow it. In this article, we describe the work in progress distributed corpus management system using a large cluster of commodity machines. The implementation is based on the Manatee corpus management system and written in the Go language. Currently, the implemented features are query evaluation, concordance building, concordance sorting and frequency distribution calculation. We evaluate the performance of the distributed system on a cluster of 65 commodity computers and compare it to the old implementation of Manatee. The performance increase for the distributed evaluation in the concordance creation task ranges from 2.4 to 69.2 compared to the old system, from 56 to 305 times for the concordance sorting task and from 27 to 614 for the frequency distribution calculation. The results show that the system scales very well.

Keywords: corpus, parallel, distributed, concordance

1. Introduction

Every year, the amount of text produced and stored increases, and so do the requirements to process and search it. Some of the largest corpora we build for use in Sketch Engine (Kilgarriff et al., 2014) now take months to compile, and their searching leaves a lot to be desired even on today's state-of-art machines, mainly due to lack of parallelization and storage bottlenecks.

Building on the approach described in (Rábara et al., 2017), where we report on our experiments in acceleration of corpus search using shared-memory multiprocessor machines, we developed an extension to the Manatee corpus management system (Rychlý, 2007) which allows us to distribute corpus operations over a cluster of commodity computers.

2. System description

The distributed system is based on the reimplementation of the Manatee corpus manager in Go.

We employ the MapReduce model, where machines in a cluster work on a local part of the data in isolation – *map* it to a partial result. These results are then propagated through the cluster and collated or *reduced* to obtain the final result. Our architecture uses multiple servers and a single client. The client schedules the work to be done, manages the servers, and handles interactions with the user. The servers are where the data is stored and where the performance intensive processing is carried out. The machines within the cluster communicate using a custom HTTP based protocol. The requests transmitted to the servers are encoded as JSON, while the much larger results are returned in a denser and more performant Protocol Buffers based format, as the encoding and decoding steps turned out to be a noticeable performance bottleneck.

2.1. Implemented features

Currently, only the most fundamental, but at the same time most difficult to implement, operations have been implemented. These are query evaluation, concordance building, sorting and frequency distribution calculations.

Corpus compilation is done in the same way as in the current Manatee system, except that each of the servers compiles its own local part of the corpus. Currently, we do not build any distributed indices and each part of the distributed corpus is a standard Manatee compatible corpus on its own. Uploading of the corpora to the servers is done out-of-band without any intervention of the system itself by standard UNIX tools.

Some important functionality has not been implemented yet, such as fault tolerance and fail-over, as these were deemed unnecessary in a proof-of concept system. Implementing some functions, such as the Word Sketch, should be straightforward, while other functions, such as thesaurus, might end up being calculated with the help of the servers, but ultimately stored on and queried by the client itself.

2.1.1. Concordance building

Evaluating Corpus Query Language queries and building the resulting concordance is done similarly as in the original implementation. In the original implementation, concordances are stored as lists of positions in the corpus. The textual form is generated on the fly when user requests a specific part of the concordance. Similarly, sorting and filtering operations manipulate numerical representations of tokens. This is not possible in the distributed implementation, as the other workers have no knowledge of lexicons and contents of the other parts of the corpus. Therefore, we build the textual representation of concordance, including the contexts, immediately on the servers. The final result is obtained by concatenating the partial results on the client.

2.1.2. Concordance sorting

It is often required for the concordance to be sorted by some criteria. We handle this case by distributed merge-sort. Partial results are generated and sorted on the servers and then streamed the client where the last merging pass is carried out. An optimization which avoids transferring all the partial results is in place for the case where a specific page of the concordance is requested. We build a sort index on each

Query	Result size	C++ implementation		Go implementation	
		asynchronous	synchronous	cluster	single machine
`[word="work.*ing"]`	3,696,606	14.62	20.04	0.99	37.19
`[word="confus.*"]`	702,436	14.88	18.24	0.29	34.89
`[word="(?i)confus.*"]`	731,452	25.44	34.51	0.76	43.63
`[lc=".*ing" & tag="VVG"]`	231,346,778	61.10	242.51	5.01	222.76
`[lemma_lc="good"] [lc="plan"]`	20,804	6.18	6.27	0.46	18.94
`[word=".*ing"]`	371,767,766	240.00	626.94	4.18	241.77
`[tag="JJ"] [lemma="plan"]`	553,724	3.18	18.21	1.33	15.74
`"some" [tag="NN"]`	5,107,984	3.28	36.14	1.46	22.72
`[lc=".*ing" & tag!="VVG"]`	141,174,215	61.75	229.08	5.84	281.31
`[tag="DT"][lc=".*ly"] [lc=".*ing"] [word="[A-Z].*"]`	54,957	334.01	more than 3600	32.88	more than 3600
`[tag="DT"][lc=".*ly"] [lc=".*ing"] [word="[A-Z].*" & tag!="P.*"]`	29,053	344.57	more than 3600	35.44	more than 3600

Table 1: Concordance building performance

Query	C++ implementation	Go implementation	
		cluster	single machine
`[word="Gauss"]`	26.89	0.48	26.85
`[word="recurrence"]`	180.16	1.09	52.00
`[word="enjoyment"]`	410.08	1.35	123.93
`[word="test"]`	492.79	3.29	158.38
`[word="said"]`	266.69	4.51	100.77
`[word="a"]`	more than 3600 s	23.99	more than 3600 s
`[word="the"]`	more than 3600 s	54.73	more than 3600 s

Table 2: Concordance sorting performance

Query	Result size	C++ implementation	Go implementation	
			cluster	single machine
`[word="Gauss"]`	497	17.01	0.36	12.80
`[word="recurrence"]`	1,580	159.32	0.33	31.90
`[word="enjoyment"]`	4,841	361.91	0.59	101.56
`[word="test"]`	33,100	482.94	3.67	138.39
`[word="said"]`	208,676	147.50	5.29	67.25
`[word="a"]`	1,700,427	576.39	15.42	136.90
`[word="the"]`	3,716,817	1273.01	28.86	621.96

Table 3: Frequency distribution performance

of the servers, which speeds up random accesses to the local sorted concordance. The client then merges the partial sort indices to obtain a global index, which can be queried to obtain the location of the requested concordance lines within the cluster. As the concordance is already in a textual form, it is not necessary to perform additional lookups of the sorting keys in the corpus. The trade-off is larger amount of data that needs to be transferred between the workers. The merging step on the client is the bottleneck of this operation. The performance is strongly affected by the transfer and decoding of the data coming from the clients, so we only retrieve the pages of the partial result necessary to display the current final result page. To speed up the decoding, we chose a dense binary-based representation built on the Protocol Buffers toolkit.

2.1.3. Frequency distribution

The frequency distribution feature is used to generate various kinds of histograms or contingency tables, such as the counts of context words which appear at a particular position with respect to a CQL query. This is a demanding operation, as it might be necessary to transfer lot of data between the server and the client. Multiple context features can be specified and the number of values of each of them can be as large as the size of the lexicon, therefore the result size can grow exponentially as additional context features are added. The frequency distribution can be obtained in shorter time with the distributed architecture, but the maximum allowable size is still limited by the available memory of the client. The partial histograms are calculated and sorted on every server and then transferred to the client, where they are combined to obtain the complete distribution. To speed up the

3. Performance evaluation

3.1. Hardware and software environment

The tests were carried out on a cluster consisting of 65 diverse computers. Each of them configured with 16 GB RAM and Ivy Bridge or Haswell Intel Core i5 4-core processor, depending on the particular machine. These machines are standard desktop computers accessible by students for general use in computer labs, so we cannot claim that the environment is free from external influences, but we found that the results during off-hours are consistent and repeatable. The operating system used was Fedora Linux.

3.2. Evaluation corpus

The corpus we used to evaluate the system is enTenTen12 from our TenTen family of Web corpora (Jakubíček et al., 2013). It is an English corpus consisting of approximately 13 billion tokens. Each of the 65 machines processed a part of the corpus with 200 million tokens stored on its local hard-disk.

3.3. Results

3.3.1. Concordance building

The Table 1 shows the time in seconds which was necessary to compute the concordance for a few selected queries from our test suite. Each of the concordance lines contains the text representation of the keyword and at most 40 characters to the left and at most 40 characters to the right, including paragraph boundaries and a document identificator.

This benchmark tests the raw speed of the mostly sequential index reads and the construction of the textual concordance representation, which utilizes the lexicons and the attribute text. These two operations are seek-intensive, but well-behaved with respect to the caching behavior.

We measured the performance of the current C++ implementation in two modes. In asynchronous mode, the time needed for the retrieval of the first 20 rows is given. This represents the time necessary to serve the first page of results to the user and approximates the latency of the system.

In synchronous mode, we waited until the whole text representation of the concordance had been constructed.

The asynchronous mode is not significantly faster compared to the synchronous mode, relative to the amount of result rows. This is because after the first 20 lines have been processed, lexicons and large parts of attribute texts are cached in system memory, so generating the rest of the result is much less expensive.

Performance of the distributed implementation was measured on the 65 worker cluster in one case and on a single worker in the second case. Asynchronous query evaluation has not been implemented in this system – the results need to be combined from the different workers as their order and location on the servers is not known beforehand, but we would still like to preserve their order in the face of indeterminism. The speedup from the distributed implementation varies from 2.4 to 69.2 when evaluated over the whole test suite.

The performance difference between the new implementation and the current implementation on a single machine is caused by optimizing the new implementation for complex queries with large result sizes, on the order of 5 % of the whole corpus. These issues have been largely eliminated in subsequent versions of the software, but we didn't have the opportunity to reevaluate the performance on the cluster yet.

On a per-processor basis, the distributed system is less efficient by design, as the lexicons present on each of the servers have significant overlap, but the total amount of memory that can be used to cache the data is much larger. As the working sets are smaller, every processor can be utilized better, as there are less cache spills.

3.3.2. Concordance sort

The Table 2 shows the time in seconds necessary to evaluate a query, sort the result and generate a textual representation of a concordance for a given query. The concordance was sorted using the three lowercased words to the right of the keyword as the key. As the performance of query evaluation has been examined in the previous section, the queries chosen for the evaluation consist of a single word to match each, as only the bounds of every match are used in the subsequent step. The structure of the query itself matters little. so We chose the words by their frequency, which is listed in the Table 4, to account for the various possible result sizes.

Query	Frequency
`[word="Gauss"]`	2,132
`[word="recurrence"]`	28,927
`[word="enjoyment"]`	157,287
`[word="test"]`	1,625,427
`[word="said"]`	10,842,497
`[word="a"]`	241,926,311
`[word="the"]`	547,226,436

Table 4: Query result sizes

The speedup ranges from 56 to 305 compared to the current implementation and from 1.0 to 3.5 when running on a single machine, excluding the operations which took too long to complete. The result for the query `[word="Gauss"]` takes almost the same time for the current and the new implementation, even though the new implementation uses multiple cores on every machine. This is because of the low-level inefficiencies of the new implementation. Compared to the concordance building benchmark, the new implementation is able to catch up because the sorting step can be more easily parallelized.

3.3.3. Frequency distribution

The Table 3 shows the time in seconds necessary to evaluate a query and generate the frequency distribution with respect to a single token immediately to the right.

The speedup between the current and new implementations ranges from 27 to 614 and from 1.4 to 5.0 when a single machine is used. The performance increases for queries with large result sets are limited by the large amount of data which needs to be transferred from the servers to the client and by the merging step carried out on it. Inserting another layer of workers to help with partial result merging between the servers and the client could help to decrease the total time necessary, at the cost of a possible increase in latency. However, we would like to avoid a stratified architecture, as it introduces large amounts of complexity and points of failure into the system.

The design of the system predates the availability of abundant cheap memory, so it behaves well under memory pressure. When only query evaluation is considered, the system is not sensitive to memory pressure – indices are processed in streaming fashion from disk, and memory is only used for caching. This extends to the distributed implementation. Frequency distribution, on the other hand, is inherently memory intensive, and using the current approach it is necessary to store the whole result on the client, which therefore needs to have the same amount of memory installed as if it were the singular machine used in the C++ implementation. In the future, it might be beneficial to split the frequency distribution functionality into more pragmatic features tailored to the specific use cases instead of the generic and elegant approach used now.

4. Conclusion

We describe the architecture of a proof-of-concept distributed corpus management system we developed and evaluated its performance on a large corpus. The results show that the performance of the distributed system scales very well and can support large scale corpus processing without the need for fancy storage arrays and distributed filesystems. Some queries that cannot currently be feasibly evaluated in interactive scenarios are now within reach, allowing more detailed analyses.

Acknowledgements

This work has been partly supported by the Ministry of Education of CR within the LINDAT-Clarin project LM2015071 and by the Grant Agency of CR within the project 18-23891S.

Jakubíček, M., Kilgarriff, A., Kovář, V., Rychlỳ, P., and Suchomel, V. (2013). The tenten corpus family. In *7th International Corpus Linguistics Conference CL*, pages 125–127.

Kilgarriff, A., Baisa, V., Bušta, J., Jakubíček, M., Kovář, V., Michelfeit, J., Rychlỳ, P., and Suchomel, V. (2014). The sketch engine: ten years on. *Lexicography*, 1(1):7–36.

Rábara, R., Rychlý, P., Herman, O., and Jakubíček, M. (2017). Accelerating corpus search using multiple cores. In Piotr Bański, et al., editors, *Proceedings of the Workshop on Challenges in the Management of Large Corpora and Big Data and Natural Language Processing (CMLC-5+BigNLP) 2017 including the papers from the Web-as-Corpus (WAC-XI) guest section*, pages 30–34, Mannheim. Institut für Deutsche Sprache.

Rychlỳ, P. (2007). Manatee/bonito-a modular corpus manager. In *1st Workshop on Recent Advances in Slavonic Natural Language Processing*, pages 65–70.