Scalable Search Platform: Improving Pipelined Query Processing for Distributed Full-Text Retrieval

Simon Jonassen
Advised by Prof. Svein Erik Bratsberg and Adj. Assoc. Prof. Øystein Torbjørnsen
Norwegian University of Science and Technology, Trondheim, Norway
{simonj, sveinbra}@idi.ntnu.no, oystein.torbjornsen@microsoft.com

ABSTRACT

In theory, term-wise partitioned indexes may provide higher throughput than document-wise partitioned. In practice, term-wise partitioning shows lacking scalability with increasing collection size and intra-query parallelism, which leads to long query latency and poor performance at low query loads. In our work, we have developed several techniques to deal with these problems. Our current results show a significant improvement over the state-of-the-art approach on a small distributed IR system, and our next objective is to evaluate the scalability of the improved approach on a large system. In this paper, we describe the relation between our work and the problem of scalability, summarize the results, limitations and challenges of our current work, and outline directions for further research.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

Keywords

distributed query processing, optimization, inverted index, scalability

1. INTRODUCTION

In the recent years query-based search became an essential component of any computer application. With a rapidly increasing information volume and access demands, IR has stepped from single-node to multi-node and even multi-site systems [7]. In this discussion a central place is given to index partitioning and query processing methods. Here, a fundamental choice is whether the collection should be partitioned by term (TP) or by document (DP).

An IR system can be evaluated by three main aspects: cost, result quality and performance. In this context, *scalability* describes how good a system is to cope with an increase in the data volume and the number of users, while maximizing the performance and result quality, and minimizing its cost. During the last 20 years many different studies have tried to evaluate the effect of the choice of partitioning method on the performance and therefore scalability of a distributed search engine. The only conclusion can be made so far is that both TP and DP can be advantageous depending on a large number of system parameters, such as the number of nodes, collection size, disk and network access characteristics and the query processing model [16].

Rather than asking whether TP or DP is more scalable, the question we address is how can we improve the scalability of TP. The main reason to look at TP is that processing of a query q containing $|q|$ query terms involves only a small number of nodes, processing $|q|$ posting lists in total (compared to $|q|$ posting lists on each of the n processing nodes with DP). As there are many different ways to process a query over a TP index, we chose to look at the pipelined query processing approach [28] (PL), where each query is processed by routing a query bundle through a set of nodes responsible for the query terms, processing the corresponding posting data and finally returning the k top-scored candidates as a result. Compared to the ad-hoc TP, PL distributes the processing task over all of the query nodes and tries to reduce the amount of processed and transferred data.

According to Büttcher *et al.* [6] the main problems of TP/PL are poor load balancing, inability to scale with increasing collections size, limited intra-query concurrency and term-at-a-time processing. Webber [36] shows that with resolved load balancing issues, TP/PL outperforms DP in terms of the maximum throughput and total I/O on a system using the GOV2 corpus distributed over 8 processing nodes. However, Webber admits that PL is inefficient at light to moderate query loads due to its sequential nature. A question that we can ask is what happens if we resolve all of the problems, will PL become an ultimate highly-scalable approach to distributed query processing?

In this paper we show that at this point we have developed techniques to deal with most of the problems of TP/PL. The task we face now is to scale the approach beyond a document collection of a few millions of documents distributed across a small number of nodes. At the same time, we discuss the further extension and complementary alternatives to our approach with a purpose to contribute to the development and understanding of scalable query processing.

1.1 Research Questions

The original description of the PhD topic is as follows: "The PhD student will focus the research activity in the area of scalable search platforms. The research challenge here is to develop methods to scale with respect to search power and data volume to be indexed, and at the same time give good performance and minimizing human management

WWW 2012 Companion, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1230-1/12/04.

during scaling or churn in the hardware resources". This description leads to three high-level research questions:

- "What is the definition of scalability?"
- "Which methods to make scalable search applications exist today and which limitations do they have?"
- "How can we extend these methods to compete with intensively increasing document collections, user populations, high result demands and frequent hardware failures?"

The choice of the pipelined query processing approach as the method of interest leads to more specific questions:

- "What are the main challenges and limitations of the pipelined query processing?"
- "How can we resolve these issues?"
- "How can we further improve the performance and scalability of the method?"

2. STATE OF THE ART

Scalability has been the topic for several books and surveys within IR. In the most recent work, Cambazoglu and Baeza-Yates [7] have structured challenges and known techniques for crawling, indexing and query processing into several systems of increasing size, while Hölzle and Barroso [14] have covered challenges and techniques for data center design and operation. In the following, we focus on the query processing aspects and review only a limited number of techniques related to our work. For a better overview we refer to [6, 7, 14].

2.1 Scalability

We define scalability of a search engine in five directions:

- query arrival rate and throughput
- query processing time and latency
- document collection size
- amount and frequency of changes to documents
- cost of the system

According to Chowdhury and Pass [10], who define three main performance indicators of a scalable search system as query response time, throughput and utilization, there are two ways to improve the performance. The first one is to perform index partitioning and replication using a large number of nodes, and the other is to speed-up the processing itself.

2.2 Partitioning and replication

For a large document collection, an ad-hoc high-performance system uses a large number of independent commodity computers [2] to process, analyse and index its content and then use the resulting index to answer any incoming query. To distribute the query processing, the nodes has to partition and replicate the underlying index and then cooperate their work within each query.

Index partitioning. Two general strategies to inverted index partitioning is to split it either by term or by document ID. During the last 20 years, variants of TP and DP have been compared by a large number of publications, which show that both methods can be advantageous depending on the system settings, query model and implementation details. In particular, [1, 16, 30] demonstrate the efficiency of TP in presence of fast network transfer, efficient pruning and/or a large number of concurrently processing queries. While [15, 16, 24] show that high network load, potential bottleneck at the ranker node, and poor load-balancing can make this method to be less preferable than DP. Finally, [8] shows that TP gives higher throughput, while DP gives shorter latency.

The method of our interest, PL was first introduced by Moffat *et al.* [28]. Different from the traditional TP, PL distributes processing over all of the query nodes and reduces processing and network load with space-limited accumulator pruning [21]. The original paper has demonstrated a significant improvement over the TP baseline, but the method was outperformed by a corresponding DP approach. According to the authors, poor load-balancing is the main problem of PL. This issue was addressed in the following work by Moffat *et al.* [27] and Webber [36]. Their final results [36] show that with a query-log-based term-assignment and replication method, PL is able to outperform TP.

Log-based posting list assignment within TP/PL was followed by Zhang *et al.* [39], who presented a graph partitioning approach to reduce the communication cost, and Lucchese *et al.* [23], who addressed the number of active servers and load imbalance as an optimization problem. In both papers, the main objective is stated as an NP-complete problem solved by offline approximations, without accounting for repartitioning cost or complexity.

Alternatively to PL, a series of publications by Marin *et al.* (e.g.[25, 26]) presents a TP approach where queries are processed by iterative retrieval a K-sized pieces of data from each fetcher node and processing these by a ranker node. The underlying inverted index is frequency-ordered. Further they suggest to apply bulk-synchronous processing (BSP) model, round-robin ranker assignment and switch between synchronous and asynchronous query processing model depending on the query load.

In a recent work, Feuerstein *et al.* [13] have presented a hybrid approach, where the index is partitioned by both terms and documents at the same time. This approach (2D) is expected to have less overhead with a large number of query processing nodes, but shorter posting lists and better load-balancing. In the following work [12], the approach was combined with the methods by Marin *et al.*

Index replication Index replicas can be used to process several queries in parallel and to handle node failures. According to Risvik [31], a combination of DP and replication results in an architecture that can scale with respect to both collection size and query rate. As an alternative to this method, Marin *et al.* [26] have recently presented an efficient approach combining clustered 2D partitioning, BSP and caching.

Multi-tier organization and static pruning Another classical method for scalable distributed indexes is to partition documents into several disjoint tiers. Processing of each query starts at the top-tier and falls down one tier at a time when necessary [31]. Similar effect can be achieved by static index pruning, where a small portion of an index can be processed before accessing a full index. Skobeltsyn *et al.* [32] have presented results on efficient pruning of both terms and documents in combination with caching.

2.3 Accelerated query processing

Alternatively to partitioning, replication and static pruning, it is possible to speed-up the processing itself. A naive solution is to order better hardware, which would increase a

system's cost rather than performance. Instead, we focus on speeding-up the processing on the existing hardware. Below we briefly review the most interesting techniques.

Index organization. As already mentioned, frequency-ordered lists can be used instead of document-ordered. Although they have been demonstrated to be highly efficient [33], document-ordered indexes are easier to maintain and combined with careful document ID assignment [37] and query processing optimizations can be efficient as well.

Compression and caching. Inverted file compression has always been a central issue for efficient search engines. Current research concentrates on super-scalar, branch- and loop-optimized, word-level bulk-compression methods. Suel *et al.* [38] have recently demonstrated efficiency of PFoR [40] compression and came up with several improvements [37].

Due to a natural skew in term popularity and presence of search trends, caching is an efficient method to improve processing load and I/O. Traditional systems look at up-to three levels of caching [22] – posting lists, intersection and result caches. A large body of work has been done in the direction of cache eviction, admission and second chance policies and static vs dynamic caching. More recent studies look at 5 cache levels [26] for distributed systems and a connection between compression and cache methods [38].

Skipping, pruning and query optimizations. Self-skipping indexes have been previously presented by several papers [3, 5, 9, 29]. Skipping has been shown to be highly efficient for conjunctive (AND) queries but is more computationally expensive for disjunctive queries (OR). Early query processing optimizations have been discussed by Turtle and Croft [35], who also presented an efficient pruning heuristic, MaxScore, which avoids processing candidates that are guaranteed to be absent from the final result set. The authors mention that this technique is highly efficient in a combination with skipping, but do not describe an exact implementation. Two other optimizations for partial evaluation, Quit and Continue, have been discussed by Zobel and Moffat [29] and later outperformed by a space-limited pruning method by Lester *et al.* [21] (the method used by PL). Compared to MaxScore, Lester's method neither guarantees the same results as a full evaluation or applies skipping, but restricts the number of maintained accumulators and expects the best results to be found among these. As another alternative, Broder *et al.* [4] have presented an efficient approach principally opposite to MaxScore, WAND. This mehtod was recently extended by Suel *et al.* [11].

Multi-core processing. Several recent publications look at multi-core techniques for IR. Most recenly, Tatikonda *et al.* [34] have studied fine-grained intra-query parallelism within posting list intersection and reported a 5.75 times speed-up on an eight-core CPU. However, their query processing model is limited to intersection of memory-based posting lists and skip-pointers used for task-scheduling are stored as non-compressed arrays.

Other methods. Techniques outside our focus include use of GPU and adaptations to the map-reduce framework, column-stores and key-value stores.

3. METHODOLOGY

The work behind the thesis can be classified as an exploratory, experimental research based on a real implementation. Originally we intended to extend an existing open-source search engine, such as Solr/Lucene, Terrier, Zettair and MG4J. We chose Terrier because of its intuitive, well-organized source code and being used by a large number of academic publications. The initial work led to writing a distributed version of Terrier 2.3. However, due to performance-oriented optimizations the most interesting features of Terrier were removed and the search engine itself was used just as an index manager. Our later work led to writing a custom inverted index. As being very lightweight and specifically optimized, it replaced Terrier in the rest of our work. Worth to mention that many of the concepts used in our final framework are inspired by Terrier, Zettair and MG4J.

For our experiments we use the 426GB TREC GOV2 corpus containing 25 mil. documents, Terabyte Track Efficiency Topics and TREC Adhoc Retrieval Topics and Relevance Judgements, and the Okapi BM-25 similarity model. The experiments are done using either a single node or a 9 node cluster interconnected with a Gigabit network.

4. PROPOSED METHODS AND RESULTS

In this section we review the methods we have developed and evaluated so far. We describe which problems they address, the idea behind, the essential results and the relation to our main problem. For a more detailed description of the methods and results we refer to the original publications.

Combined semi-pipelined processing

In our first paper [17], we have addressed four observations of the state-of-the-art PL method: First, non-parallel disk-accesses and bundle processing result in long query latencies. Second, accumulators have less compression potential than inverted list postings. Third, PL is not always better than a non-pipelined processing, where a single node receives and processes complete compressed posting lists. Fourth, the routing strategy does not minimize the number of transferred accumulators.

To resolve these issues, we presented an approach combining three different techniques. First, a semi-pipelined query processing – a combination of parallel disk-access and pipelined processing, which is done by sending a bundle replica to each of the query nodes instead of just the first node. Each node receiving the bundle fetches and decompresses all posting data as soon as possible, but the following query processing is done in a pipelined way. Second, we proposed a decision heuristic to choose between semi- and non-pipelined execution based on the estimated amount of data to be transferred for each query. And third, we suggested to route the query by the increasing longest posting list length instead of the increasing smallest collection frequency.

According to the experimental results, the method combines the advantages of the pipelined and non-pipelined query processing and outperforms both methods in latency and throughput. However, the query processing model requires posting lists to be fetched and decompressed completely, which is similar to the original implementation of PL [36], but additionally it requires the posting lists in each subquery to be present in the main memory at the same time. This results in a large memory footprint. Finally, the bitwise compression methods for posting data (gamma coding of d-gaps and unary coding of frequencies) can be criticized for being inefficient for a performance-oriented search engine. These results led us to an investigation of compression-efficient pruning methods for pipelined query processing.

Efficient compressed self-skipping index

In our second paper [18], we have addressed efficient skipping for disjunctive queries on a monolithic system. Because of a large performance gap, practical search engines have traditionally processed queries in a conjunctive (AND) rather than disjunctive (OR) mode. Further, the previously presented self-skipping indexes [3, 9, 29] do not account for block-wise disk-access and bulk-compression methods. Here, we proposed a new self-skipping inverted index designed specifically for modern super-scalar bulk-compression methods (such as NewPFor[37]), and presented an updated version of MaxScore (including a complete algorithmic description) and a skipping-adapted version of the pruning method by Lester *et al.* From our results, both query processing methods significantly reduce the number of processed elements and reduce the average query latency by more than three times and significantly reduce the performance gap between AND and OR queries.

Improving TP/PL with skipping

In the following work [20], we adaptated our skipping methods to a distributed system applying TP/PL. The main problems addressed in this work are the poor scalability of PL with respect to increasing collection size and the performance gap between AND and OR queries. Here, we have presented a skipping-adapted version of the original PL approach applying Lester's pruning method, and a novel skip-optimized PL approach applying MaxScore heuristics. Additionally, we suggested an alternative posting-list assignment method, by MaxScore. From the results, our methods significantly improve the latency and throughput, and eliminate the gap between AND and OR queries. Finally, both MaxScore PL technique, which introduces document-at-a-time (DAAT) processing of sub-queries, and the MaxScore partitioning lead to several possible optimizations, which we are going to explain below. These include intra-query concurrent processing, a hybrid combination with non-pipelined query processing and possibility for dynamic load balancing.

Intra-query concurrent TP/PL

In the most recent paper [19], we have addressed the lack of intra-query parallelism within PL by a further extension of the pipelined MaxScore method from the last work. For each query we suggested to divide the document ID space into several sub-ranges, which we call fragments. Because of DAAT sub-query processing, as soon a fragment is fully processed on one node, it is possible to send the corresponding accumulators to the next node, and therefore overlap the execution of the same query on different nodes. Further, at lower concurrency levels, different fragments of the same sub-query can be processed concurrently on the same node by maintaining a small number of executors associated with each particular query and sharing a common pruning threshold variable. Our experimental results indicate that the final approach reaches a throughput similar to the previous method at about half of the latency, and on the single query case the latency improvement is up to 2.6 times.

5. FURTHER WORK

Besides the presented ideas and results, our work had introduced many new challenges and opportunities that can be addressed both during the rest of this PhD work and far beyond the dissertation time. We organize the further work in three main directions: a) evaluation of the scalability, b) further extension of the methods, and finally, c) addressing open problems and considering alternative solutions.

5.1 Limitations of our current results

Our results indicate a significant performance improvement over the state-of-the-art method. However, as the experiments are done on a fixed system of 8 processing nodes and a relatively small document collection, they do not evaluate the scalability itself. Therefore, further work has to include an evaluation on a bigger cluster and using a significantly larger document collection. Further, we see a great value in the evaluation of a system where both the number of nodes and the document content change dynamically.

5.2 Further extension of our methods

With the MaxScore term assignment strategy, posting lists are sorted by the MaxScore value and then divided between the nodes, so node i receives the lists with the maximum score above those stored by node $i+1$ but below $i-1$. This provides several opportunities for further improvements [20]. First, we can consider a dynamic load-balancing method where some of the posting lists can be moved to a neighbouring node in order to reduce the load on the current node. In order to minimize the network and I/O load, two consecutive nodes may share some of the posting lists around the split value. Second, as the posting list size is correlated to the maximum score, the posting lists stored on the first few nodes are quite short. Therefore, we can consider a hybrid strategy where for each query some of the nodes may only read and transfer its posting lists to a node later in the query route. The receiving node is then the first node in the query pipeline. This can further be extended by caching remote posting lists on the receiving node. Third, we can also consider a multi-tier system where the first tier stores only the most relevant terms and the last tier stores the least relevant terms. Thus, query processing involves the last tier only when the results returned by the primary tier are not sufficient.

Within each node further improvement can be done by caching, enhancing the skipping index and the processing method, and finally optimizing for multi-core processing.

5.3 Alternative methods and open problems

Our current methods build on a document-ordered index and apply the MaxScore heuristic. Alternatively, the pipelined approach could also be adapted to the techniques by Marin *et al.*, Suel *et al.* [11] and Tatikonda *et al.* [34]. However, at the present moment it is unclear whether impact- or frequency-ordered indexes are significantly better than carefully optimized document-ordered indexes, multi-level WAND is any better than MaxScore, or intersection-first approach is more efficient than early termination/pruning.

6. CONCLUSIONS

We have contributed to resolving some of the most challenging problems of the pipelined query processing in order to make this approach truly scalable and highly efficient. Our current results indicate a significant improvement over the state-of-the-art approach. However, at this point, we have not evaluated the scalability itself. Future work can be done in the direction of a further extension of our methods, scalability evaluation and addressing the open problems.

Acknowledgement. This work is supported by the iAd Centre and funded by the NTNU and the Research Council of Norway.

7. REFERENCES

[1] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, 2001.

[2] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2), 2003.

[3] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *SPIRE*, 2005.

[4] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, 2003.

[5] S. Büttcher and C. Clarke. Index compression is good, especially for random access. In *CIKM*, 2007.

[6] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010.

[7] B. B. Cambazoglu and R. Baeza-Yates. Scalability challenges in web search engines. In *Adv. Topics in Inf. Retr.*, volume 33. 2011.

[8] B. B. Cambazoglu, A. Catal, and C. Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In *ISCIS*, volume 4263, 2006.

[9] F. Chierichetti, S. Lattanzi, F. Mari, and A. Panconesi. On placing skips optimally in expectation. In *WSDM*, 2008.

[10] A. Chowdhury and G. Pass. Operational requirements for scalable search systems. In *CIKM*, 2003.

[11] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *SIGIR*, 2011.

[12] E. Feuerstein, V. Gil-Costa, M. Mizrahi, and M. Marin. Performance evaluation of improved web search algorithms. In *VECPAR*, 2011.

[13] E. Feuerstein, M. Marin, M. Mizrahi, V. Gil-Costa, and R. Baeza-Yates. Two-dimensional distributed inverted files. In *SPIRE*, 2009.

[14] U. Hölzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[15] B. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2), 1995.

[16] S. Jonassen and S. E. Bratsberg. Impact of the Query Model and System Settings on Performance of Distributed Inverted Indexes. In *NIK*, 2009.

[17] S. Jonassen and S. E. Bratsberg. A combined semi-pipelined query processing architecture for distributed full-text retrieval. In *WISE*, 2010.

[18] S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *ECIR*, 2011.

[19] S. Jonassen and S. E. Bratsberg. Intra-query concurrent pipelined processing for distributed full-text retrieval. In *ECIR*, 2012.

[20] S. Jonassen, Ø. Torbjørnsen, and S. E. Bratsberg. Improving the performance of pipelined query processing with skipping. *unpublished work*.

[21] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, 2005.

[22] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, 2005.

[23] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *InfoScale*, 2007.

[24] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In *SPIRE*, 2000.

[25] M. Marin and V. Gil-Costa. High-performance distributed inverted files. In *CIKM*, 2007.

[26] M. Marin, V. Gil-Costa, and C. Gomez-Pantoja. New caching techniques for web search engines. In *HPDC*, 2010.

[27] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, 2006.

[28] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 2007.

[29] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 1996.

[30] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *DL*, 1998.

[31] K. M. Risvik. *Scaling Internet Search Engines - Methods and Analysis*. PhD thesis, 2004.

[32] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates. Resin: a combination of results caching and index pruning for high-performance web search engines. In *SIGIR*, 2008.

[33] T. Strohman and W. Croft. Efficient document retrieval in main memory. In *SIGIR*, 2007.

[34] S. Tatikonda, B. B. Cambazoglu, and F. P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, 2011.

[35] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 1995.

[36] W. Webber. Design and evaluation of a pipelined distributed information retrieval architecture. Master's thesis, 2007.

[37] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, 2009.

[38] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, 2008.

[39] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. *Paral. and Dist. Proc. Symp., Int.*, 2007.

[40] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.