# Improving Throughput of a Pipeline Model Indexer

Matt Crane
Department of Computer Science
University of Otago
Dunedin, New Zealand
mcrane@cs.otago.ac.nz

Andrew Trotman
eBay Inc.
atrotman@ebay.com

David Eyers
Department of Computer Science
University of Otago
Dunedin, New Zealand
dme@cs.otago.ac.nz

## ABSTRACT

There are many competing models for the indexing process of an information retrieval system, one of which is a pipeline based model. Information retrieval is also an inherently parallel process, indexing one document is independent of another document. A pipeline model allows for easy experimentation on the parallelism within an indexer. In this paper we investigate areas within a pipeline where indexing throughput can be increased, as well as exploiting the inherent parallelism of indexing.

## Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software – Performance evaluation (efficiency and effectiveness)

## General Terms

Indexing, Performance

## Keywords

Buffering, Parallelism, Indexing

## 1. INTRODUCTION

There are several competing methodologies for performing indexing, one of which is a pipeline-based model. In a pipeline model a series of stages are constructed. Each of these stages is responsible for performing some operation on data that flows through it. Naturally the time spent within each stage varies, and some will include more computationally intensive computation than others. This leads to our first research question: *"Which stages of the pipeline are bottlenecks?"* After we identify these stages our next research question is: *"What can be done to improve throughput in those stages?"*

A pipeline structure also allows for easy experimentation on the inherently parallel aspects of the indexing process. Indexing one document is not affected by the indexing of another, and these can be done concurrently. This leads to our final research question: *"How much parallelism can be exploited by the indexing process?"*

*ADCS,* December 08 - 09, 2015, Parramatta, NSW, Australia

ACM 978-1-4503-4040-3/15/12 ...$15.00.
DOI: http://dx.doi.org/10.1145/2838931.2838943.

## 2. RELATED WORK

Ribeiro-Neto *et al.* [10] present three algorithms for the construction of an index in a distributed platform. In the Local-Local ($LL$) algorithm an index is created on each node and merged together afterwards. The Local-Remote ($LR$) algorithm constructs postings lists locally to each machine and distributes these to remote machines. Finally, the Remote-Remote ($RR$) algorithm sends postings to a remote machine periodically during indexing.

Melink *et al.* [9] show that a software pipeline system improves the parallelism within an indexer node in a distributed system and identify three phases that can run independently. They suggest that three pipelines provides the best performance, as each of the phases they identify can be run concurrently without causing resource conflicts.

The first of these stages is loading, where data is read from disk. The second stage described is a processing stage, in which sorted posting lists are generated. In the final stage, flushing, the generated postings lists are written to disk. Melink *et al.* found that an indexer that did not use pipelining was about 30–40% slower than using three pipelines.

Another alternative to these pipeline-based systems are MapReduce-based systems. Indeed, the introduction of the MapReduce framework to the public used indexing as an example application [1]. McCreadie *et al.* [8] showed how the single-pass algorithm of Heinz and Zobel [3] could be adapted to the MapReduce framework, and in later work provide an extensive overview and analysis of a number of different approaches to MapReduce-based indexing [7].

A more recent approach based on MapReduce has been to avoid constructing an inverted index entirely, and simply perform a linear-scan of the collection [2, 5, 6]. For some types of experimentation, such as ranking function parameter tuning [4], or searching across small numbers of queries [2], this approach performs faster than constructing an index.

Wei and Jaja [12] describe a system that parses groups of documents into a set of hybrid trie and B-Tree structures. These structures are then indexed in parallel to produce a final index. They found there is a trade-off between the number of parsers and indexers on a single node. Their distributed system showed higher throughput than alternative MapReduce indexers.

| Collection | Documents | Size |
|---|---|---|
| DOTGOV2 (.GOV2) | 25,205,179 | 426GB |
| ClueWeb09 Category B (CW09B) | 50,220,423 | 1.5TB |

**Table 1: Summary of document collections used.**

# 3. THE ATIRE PIPELINE

ATIRE's indexing process uses a producer/consumer-inspired pull-model pipeline-based structure [11]. A preprocessing pipeline is built from a series of stages, each of which performs a single process on the data that is passed through. Each stage is also built to respond to, and construct, requests for more data. The pipeline produces a stream of document objects to be handled by the indexing subsystem.

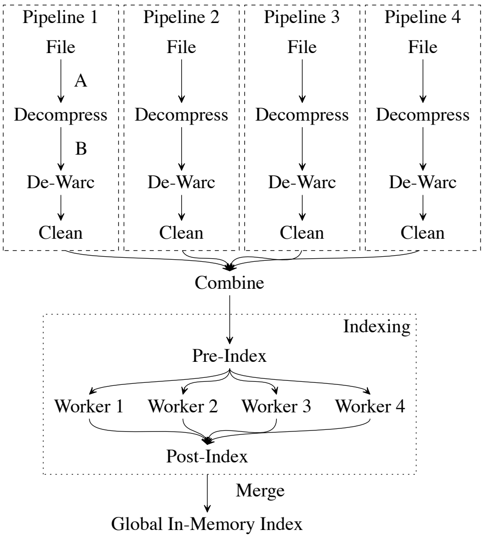


**Figure 1: An ATIRE pipeline construction for the CW09B collection. Four pipelines (grouped by dashed outlines); and internals of the pre-indexing stage (grouped by the dotted line), are shown.**

An example pipeline for processing CW09B is shown in Figure 1. Shown are four pipelines each consisting of *file*, *decompress*, *de-warc*, and *clean* stages. The internals of the pre-indexing stage are shown, with four pre-indexing workers. The number of pipelines and workers created is controlled at indexing time, as discussed in Section 7.

We perform an investigation into the indexing process used within ATIRE to identify those parts of the pipeline that are decreasing the overall throughput. We perform a selection of experiments that attempt to address these areas and show an improvement in indexing throughput.

We use two document collections, details of which are shown in Table 1. Experiments were conducted without serialising the index to disk using revision `233a24b694f0` of ATIRE. Experiments were performed on single machines with Intel Xeon E5 CPUs (specifically an 8 core 2665 for CW09B, and a 4 core 2609 for .GOV2), 256GiB memory and running Linux kernel version 2.6.32.

# 4. BASELINE

A number of open-source search engine systems participated in the RIGOR reproducibility challenge at SIGIR 2015. For the challenge authors of each system prepared scripts to index and search the .GOV2 collection. The scripts were then run on the same machine (an r3.4xlarge Amazon EC2 instance), and the resulting indexing times are shown in Table 2. Each system could be configured in multiple ways, and the results presented here are the fastest among these,[1] and include serialisation time. All systems except Terrier used parallel indexing.

| System | | Indexing Time (minutes) |
|---|---|---|
| ATIRE | | 46 |
| ATIRE Quantized | | 56 |
| Lucene | `lucene.apache.org` | 85 |
| MG4J | `mg4j.di.unimi.it` | 85 |
| Galago | `lemurproject.org/galago.php` | 392 |
| Indri | `lemurproject.org/indri/` | 460 |
| Terrier | `terrier.org` | 484 |

**Table 2: Comparative time to index .GOV2 on the same hardware for a selection of open source information retrieval systems.**

# 5. WAITING

The first step of our investigation is an exploration of the stages of the pipeline that are causing delays. Examination of time spent in each stage does not immediately reveal which are causing processing delays. It appears as though an equal proportion of time was being spent in the *de-warc* and *decompress* stages. However, the *de-warc* stage is processing the decompressed, and thus significantly larger, data. With this knowledge, the *decompress* stage is the primary target to improve throughput. However, ATIRE uses standard open-source decompression routines that have already been heavily optimised, such as `zlib`. This means improvements have to be made outside of the decompression itself.

# 6. BUFFERING

Buffering is a technique that has been applied in numerous areas, for example, operating systems, networking, and graphics. Buffers allow fast parts of a program to continue processing without waiting for slow parts. A buffer after the disk reading and decompression pipeline stages should improve throughput. As buffers are constructed separately for each pipeline, there is no need to coordinate access to them.

There are three main factors relating to buffers: position, type, and size. Placing buffers in unnecessary positions increases processing time as data is moved through the buffer. There are different types of buffer: double buffers, for instance, use a secondary buffer that fills in the background while processing continues on the primary buffer. The size of the buffer is also important—taken to one extreme, a buffer of one byte is practically no different to not buffering.

## 6.1 Position and Type

As previously mentioned, both the position and type of a buffer can be important factors. In these experiments we only consider double and single buffers—triple buffers can be simulated by using two double buffers one after the other.

[1] Complete results can be found at `github.com/lintool/IR-Reproducibility`

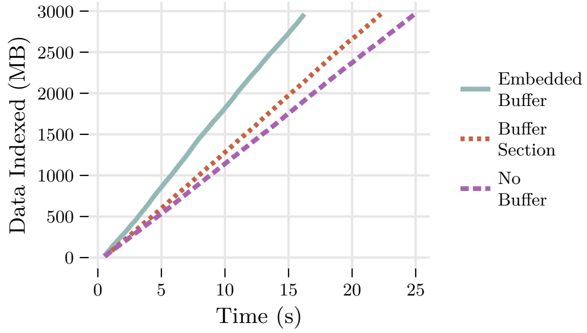
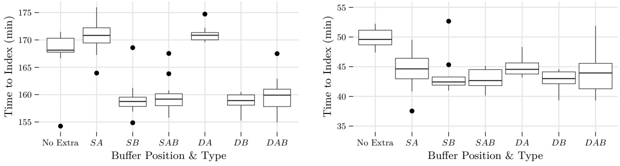


**Figure 2: Effect on indexing throughput on CW09B of embedding a 1MB buffer in decompression, an external 16MB buffer, and no buffering. Shown is the median of three runs.**

Within the pipeline structure described in Section 3 there are two places that buffers can be reasonably inserted: after the disk reading stage ($A$), and after the decompressor ($B$). Both are good places as both preceding stages are generally slow compared to the rest of the indexing process. Both positions are labelled in Pipeline 1 of Figure 1.

An alternative to an external buffer in position $A$ is embedding a buffer inside the decompression stage. Figure 2 shows the effect on indexing throughput of the different options. The buffer section results in lower throughput when compared to the embedded buffer. This is due to the overhead of the decompressor repeatedly requesting the next chunk to decompress from the buffer. Having no buffering at all shows the lowest throughput. For the remainder of this paper the embedded buffer is used.



(a) CW09B.

(b) GOV2.

**Figure 3: Effect of buffer type and position on indexing time. Shown are results from nine runs.**

Figure 3 shows the effect of buffers of different types (single and double) in the proposed locations on the indexing time of CW09B (3a), and .GOV2 (3b). The type is identified by the first letter, and positions placed in by the remaining letters. For example, $SAB$ indicates a $s$ingle buffer was placed at both positions $A$ and $B$.

Placing a buffer at $A$ decreases the throughput on CW09B. This is because, on this collection, the extra buffer introduces overheads when combined with the embedded buffer in the decompressor. Meanwhile on .GOV2 the buffer improves throughput by ≈10% (two-tailed $t$-test, $p \ll 0.01$).

The difference between the collections is likely due to the number of pipelines constructed. By default, 16 pipelines are created for CW09B, and 273 are created for .GOV2. Due to the number of pipelines, it is more likely during indexing .GOV2 that pipelines will be stalled waiting for the document indexes to be merged with the final index. By default there are eight indexing workers, and a pipeline is created for each of the top-level folders in a collection (see Section 7).

The buffer in position $B$ significantly increases throughput compared to no buffering on both collections. CW09B by ≈8% (two-tailed $t$-test, $p \ll 0.01$), and .GOV2 by ≈15% (two-tailed $t$-test, $p \ll 0.01$).

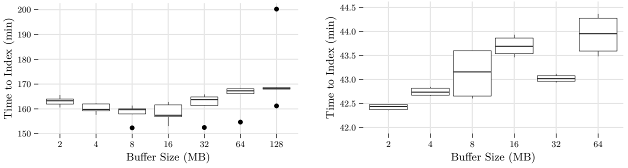
While a buffer at $B$ improved throughput on CW09B, and in either location on .GOV2, placing a buffer in both positions showed no gain over the individual positions.

Double buffering provides no gain over single buffering for either collection. One cause of this could be that the stages after buffering are substantially faster than those prior. Once the later stages had processed the contents of the first buffer, the secondary buffer is only filled a small amount. These buffers would then wait for the same time as a single buffer. The additional cost to manage the buffer then results in a lower throughput.

Further experiments are performed with a buffer in position $B$ alongside the embedded buffer in decompression.

## 6.2 Size

The third factor in relation to buffering is the size of the buffer. If the buffer is too large the later stages have to wait for it to fill. Too small and the later stages are able to process the data quickly and negate the presence of the buffer. As the size of the buffer approaches zero, the throughput approaches that of an unbuffered process with an overhead of managing the buffer.



(a) CW09B.

(b) .GOV2.

**Figure 4: Effect of buffer size on indexing time. Shown are results from nine runs.**

Figure 4 shows the effect that buffer size has on indexing time for CW09B (4a), and .GOV2 (4b). The results on .GOV2 show that the best performing size is 2MB and trends toward a smaller buffer. However, a size of zero is equivalent to no buffering, shown previously to have a lower throughput.

The results for CW09B, however, show an evident $u$-shape with the best performing tested size being 16MB. As this is the larger collection, and the size has a larger impact on indexing time (≈ 4% on CW09B compared to ≈ 1% of a 2MB on .GOV), a size of 16MB is selected for further experiments.

# 7. PARALLELISM

Indexing is inherently parallel: the result of indexing one document does not rely on the indexing of another. There are two places within ATIRE's indexing system that can take advantage of this. The first is the number of pipelines, and the second is the number of indexing workers created.

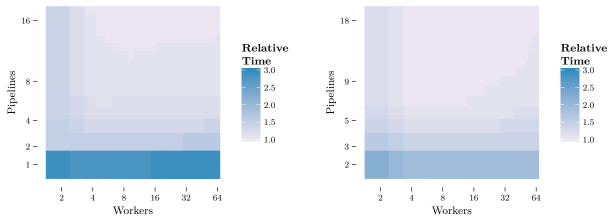
In ATIRE a separate pipeline is created for each indexing argument specified on the command line. Each pipeline is constructed in a separate thread, introducing a practical limit to the number that can be created.

Each indexing worker is also created in a separate thread, again introducing a practical limit on how many can be created. Creating too many would cause merging of the per document indexes into the global index to become a bottleneck. However, too few and the pipelines are waiting for the workers to consume the documents generated.

To investigate the effect of the number of pipelines on indexing, groupings of both collections were created, although only the process for CW09B is explained. The CW09B collection ships documents organised into compressed `warc` files, containing about 100MB of compressed content. These are grouped into folders of 100 files each, and these are then grouped into 16 top-level folders.

A grouping is created by combining top-level folders in pairs by creating hard links to the files within. Using a hard link minimises disk effects as the files remain in the same physical location. These groups are repeatedly joined in pairs until there a single group remains.

Previous experiments in this paper have used 16 pipelines for CW09B, and 273 for .GOV2, both in line with the collection as shipped. In addition, previous experiments on both collections were performed with eight indexing workers.



**(a) CW09B. Default: 16 pipelines, 8 workers.**

**(b) .GOV2. Default: 273 pipelines, 8 workers.**

**Figure 5: Relative performance (to given default) of worker/pipeline numbers.**

Figure 5 shows the relative time taken to index the collections relative to the default combination for CW09B (5a), and .GOV2 (5b). A lighter shade indicates higher throughput. The results suggest there is minimal gain to be had from further dividing collections compared to other potential improvements. Previous work has shown that taken to the extreme of storing each document in separate uncompressed files throughput drops dramatically [11].

There is a distinct drop-off as the number of pipelines is reduced. However, there is only slight variation in relative time based on the number of indexing workers for a given number of pipelines. This suggests that the work of indexing an individual document only contributes slightly to the overall time taken. As more workers are added, the throughput can be decreased (evident in Figure 5b), as either merging document indexes becomes a bottleneck, or documents cannot be produced fast enough from the pipelines. Increasing the number of pipelines alleviates this, suggesting the latter.

## 8. CONCLUSION

Section 6 shows the effect that buffering has on indexing throughput. Specifically the use of an embedded buffer within decompression substantially improves throughput. The addition of further buffering improved throughput when placed after decompression on both tested collections. Double buffering provided no additional gain. The buffer size was investigated next and there was an evident trade-off between this and indexing throughput. Providing exclusive access to the file system also increased throughput.

In Section 7 the relationship between the number of pipelines and indexing workers was investigated. There was a distinct relationship between the number of pipelines created and the total indexing time. However the best configuration was in line with the collection as shipped. The number of indexing workers also influenced the indexing time, and more workers may actually decrease throughput as merging document indexes becomes a bottleneck, or documents are not produced fast enough by the pipelines.

Results from these experiments were incorporated in the script submitted as part of the RIGOR challenge (discussed in Section 4). The results of the challenge show ATIRE has an indexing throughput that is almost twice that of the next fastest system.

## 9. ACKNOWLEDGEMENTS

The authors would like to thank the Queensland University of Technology for providing use of the machines to perform experimentation.

## References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113.

[2] T. Elsayed, F. Ture, and J. Lin. Brute-Force Approaches to Batch Retrieval: Scalable Indexing with MapReduce, or Why Bother? Technical report, HCIL-2010-23, University of Maryland.

[3] S. Heinz and J. Zobel. Efficient Single-Pass Index Construction for Text Databases. *JASIST*, 54(8):713–729.

[4] D. Hiemstra. Personal communication, 2013.

[5] D. Hiemstra and C. Hauff. MapReduce for Information Retrieval Evaluation: "Let's Quickly Test This on 12 TB of Data". In *Multilingual and Multimodal Information Access Evaluation*, pages 64–69. 2010.

[6] D. Hiemstra and C. Hauff. MIREX: MapReduce Information Retrieval Experiments, 2010.

[7] R. McCreadie, C. Macdonald, and I. Ounis. MapReduce Indexing Strategies: Studying Scalability and Efficiency. *IP&M*, 48(5):873–888.

[8] R. M. C. McCreadie, C. Macdonald, and I. Ounis. On Single-Pass Indexing with MapReduce. SIGIR '09, pages 742–743.

[9] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a Distributed Full-text Index for the Web. *ACM Trans. Inf. Syst.*, 19(3):217–241.

[10] B. Ribeiro-Neto, E. S. Moura, M. S. Neubert, and N. Ziviani. Efficient Distributed Algorithms to Build Inverted Files. SIGIR '99, pages 105–112.

[11] A. Trotman, X.-F. Jia, and M. Crane. Towards an Efficient and Effective Search Engine. In *SIGIR 2012 OSIR Workshop*, pages 40–47.

[12] Z. Wei and J. Jaja. An Optimized High-Throughput Strategy for Constructing Inverted Files. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2033–2044, Nov 2012.