# Intra-Query Concurrent Pipelined Processing For Distributed Full-Text Retrieval

Simon Jonassen and Svein Erik Bratsberg

Norwegian University of Science and Technology, Trondheim, Norway
{simonj,sveinbra}@idi.ntnu.no

**Abstract.** Pipelined query processing over a term-wise distributed inverted index has superior throughput at high query multiprogramming levels. However, due to long query latencies this approach is inefficient at lower levels. In this paper we explore two types of intra-query parallelism within the pipelined approach, parallel execution of a query on different nodes and concurrent execution on the same node. According to the experimental results, our approach reaches the throughput of the state-of-the-art method at about half of the latency. On the single query case the observed latency improvement is up to 2.6 times.

## 1 Introduction

With a rapid growth of document collections and availability of cheap commodity workstations, distributed indexing and query processing became the most important approach to large-scale, high-performance information retrieval. Two underlying, fundamentally different methods are term-wise and document-wise partitioning. With a large number of controversial comparisons and several hybrid methods presented throughout the last 20 years, both methods have their advantages and challenges. Term-wise partitioning, which we address in this paper, reduces the number of disk seeks [13] and improves inter-query concurrency [1].

The traditional query processing approach to a term-wise distributed index is to use one of the nodes as a ranker node and the remaining nodes as fetchers. With document-ordered inverted files this leads to a high network load and a large ranker overhead. In order to overcome these problems, pipelined query processing [12] suggests to create a query bundle, which includes the query itself and a set of partially scored documents, accumulators, and route it through the nodes hosting the posting lists associated with the query. At each node the accumulator structure is modified with more posting data, and at the last node the $k$ top-scored accumulators are selected, sorted and returned as a final result. Fig. 1(b) illustrates the execution of a single query.

With several optimizations [11, 17] to load balancing and replication of the most load consuming posting lists, pipelined approach has been shown to outperform document-wise partitioning in terms of query throughput. However, this comes at a cost of long query latency. Fig. 1(a) reconstructs the results presented by Webber [17], one of the authors of pipelined approach, who provided a detailed description of the methods and experiments done in their work.

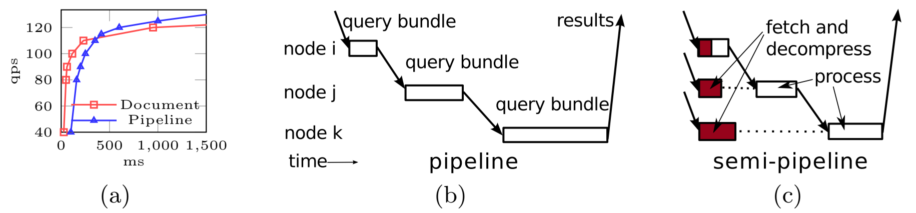


**Fig. 1.** (a) Reconstruction of the latency/throughput with varied multiprogramming levels presented by Webber [17]. (b) Pipelined and (c) semi-pipelined query processing.

The figure shows that the method has a high maximum throughput, but it is less efficient at lower query multiprogramming levels (i.e. when the number of queries processed concurrently is small). For a user-oriented search application it is important to keep the latency low and be efficient at both high and low query loads. In this paper we look at the intra-query parallelism possible with a modification to pipelined approach. Our objective is to reduce query latency at lower multiprogramming levels, while keeping the performance degradation at higher levels minimal. As the main goal we want to achieve the same throughput at a significantly lower latency.

Our contribution is as follows. We address the intra-query concurrency problem of pipelined query processing. We present a novel technique that exploits intra-query parallelism between and within each node. We evaluate our experiments on a real distributed system, using a relatively large document collection and a real query set. Finally, we suggest several directions for the future work.

## 2 Related work

Performance comparison studies of the document- and term-wise partitioning methods have been presented in a large number of publications. In this paper we refer only to a few of them [1, 4, 9, 11–13, 18]. Several works [4, 18] have also presented and evaluated hybrid partitioning and query processing strategies. Early optimizations of distributed query processing considered conjunctive (AND) queries and using the shortest term to eliminate unnecessary postings [15]. Several other studies, such as the work done by Marin *et al.* [9, 10], have used impact- or frequency-ordered lists. In our work, we look at disjunctive (OR) queries and document-ordered indexes. While impact-ordered indexes offer highly-efficient query processing [14], document-ordered indexes combined with careful document ID ordering [19], skipping and MaxScore [6, 16] or WAND [3] style processing are highly-efficient as well. At the same time, document-ordered indexes are easier to maintain and process.

Pipelined query processing (P) was presented by Moffat *et al.* [11, 12] and Webber [17]. According to the original paper [12] this method significantly improves throughput, but struggles with load imbalance and high network load. Several query-log-based term-assignment methods have been suggested in order to improve load-balancing [11, 17], reduce communication cost [20], or both [8]. The results presented by Webber [17] show that, due to inter-query parallelism, P succeeds to achieve higher throughput than document-wise partitioning once the load balancing issues are resolved. However, the author admits lacking intra-query parallelism, resulting in a poor performance under light-to-moderate workloads, and suggested that preloading of inverted lists would enable disk parallelism and therefore improve the performance.

According to Büttcher *et al.* [2], two other problems of P lie in Term-at-a-Time (TAAT) processing and a poor scalability with collection growth. The original implementation of P uses the space-limited pruning method by Lester *et al.* [7], which allows to restrict the number of transferred accumulators (thus reducing the network and processing load), but requires a complete, TAAT processing of posting data.

In our recent work [5] we have presented a combination of parallel posting list prefetching and decompression and pipelined query processing, called semi-pipelined query processing. We illustrate this approach in Fig. 1(c). Additionally, our previous work included execution of some of the queries in a traditional, non-pipelined way, and a different query-routing strategy. The results reported 32% latency improvement, but the underlying model assumed that each posting list is read and decompressed *completely* and *at once*.

As the baseline for the current work we use a modification of P applying inverted index skipping, MaxScore pruning and Document-at-a-Time (DAAT) processing within each sub-query, which we briefly describe in the next section. The intention behind this is to tackle the issues addressed by Büttcher *et al.*. The underlying query processing on each node and the index structure itself are similar to those presented in the recent work on inverted index skipping [6]. While our baseline processes different posting lists in parallel (DAAT), the bundle itself is processed by one node at a time, which is the main reason for long query latencies and a poor performance at the low query multiprogramming levels. As inverted index skipping makes semi-pipelined processing impossible, we suggest that intra-query parallelism between different nodes and within each posting list is the best way to improve the query processing performance. Finally, as we substitute space-limited pruning [7] with MaxScore, the quality of query results is equivalent to a full, disjunctive query evaluation.

## 3 Preliminaries

For a given document collection $D$ and a query $q$, we look at the problem of finding the $k$ top-ranked documents according to a similarity score $sim(d, q) = \sum_{t \in q} sim(f_{d,t}, D, q)$. Here, $sim(f_{d,t}, D, q)$ or simply $s_{d,t}$ is a term-similarity function, such as Okapi BM-25 or TF×IDF, and $f_{d,t}$ is the number of occurrences of the term $t$ in the document $d$.

**Skipping.** For any term $t$ in the inverted index, the posting list $I_t$ contains a sequence of document IDs and corresponding frequencies. Within each list postings are ordered by document ID, divided into groups of 128 entries (chunks) and compressed with NewPFor [19] using gap-coded document IDs. A hierarchy of skipping pointers, which are also gap-coded and compressed in chunks, is built on top. The logical tree is then written to disk as a prefix-traverse. A posting list iterator accessing the resulting index reads the data block-wise and keeps one chunk from each level (decompressed) in the main memory. The combination of bulk-compression, reuse of the decompressed data, index layout and buffering results in highly efficient query processing.

**Distributed index.** In order to create a distributed, term-wise partitioned index we sort posting lists by their decreasing maximum scores $\hat{s}_t = \max_{d \in I_t}(s_{t,d})$. Then we assign them to $n$ different worker nodes in a such way that the node $i$ receives the posting lists with $\hat{s}_t$ higher than those received by the node $i+1$, but lower than $i-1$, and the partitions have nearly the same size. Additional data structures such as a small local lexicon and a replica of a short document dictionary are stored on each node. An additional node $n+1$, which serves as a query broker, stores a full document dictionary and a global lexicon. During the query processing, the only structure accessed from disk is the inverted index, all the other structures are kept in the main memory as sorted arrays and accessed by binary search.

**Query processing.** At query time, each query is received, tokenized, stop-word processed and stemmed by the query broker. The resulting terms are checked in the global lexicon and the collection-based $\hat{s}_t$ values are adjusted with the normalized number of occurrences in the query. Further, the query is divided into a number of sub-queries, each containing only the terms assigned to the particular node, and a route is chosen by decreasing maximum $\hat{s}_t$ in each sub-query. Finally, the broker generates a bundle message and sends it to the first node in the route.

The bundle is processed by one node at a time. Each node in the route receives the bundle, decompresses the received accumulator set, matches it against its own posting data, generates a new accumulator set, compresses and transfers it to the next node. Query processing on each node is similar to the traditional DAAT MaxScore [16], except that it is limited only to the received accumulator set and the query-related posting lists stored on this node. Therefore, it operates with a pruning score threshold $v = \text{minHeap.min} - r$. Here, $r$ is the accumulated maximum score of the terms in the remaining sub-queries and minHeap.min is the smallest score within the $k$ top-scored results seen so far (monitored with a heap). Any partially scored accumulator can be pruned at any time if its estimated full score falls below the current value of $v$. Additionally, some of the posting lists cannot create new accumulators and therefore can be processed in a skip-mode. Accumulators that cannot be pruned have to be transferred to the next node. As $v$ increases during processing, more posting data can be skipped and more existing accumulators can be eliminated. The last node in the route does not have to create a new accumulator set. Instead, it uses only the candidate heap and when processing is done, it extracts, sorts and returns the final candidates to the broker as the result set.

Accumulators that pass the threshold are placed into a new accumulator set. Since $v$ increases within each sub-query, the accumulators in the beginning of the set may have scores below the final value of $v$, $v_{\text{final}}$. We call these false positives. In order to eliminate them, when $v_{\text{start}} < v_{\text{final}}$, an additional pass through the accumulator set is done in order to preserve only those having scores $s \geq v_{\text{final}}$. $v_{\text{final}}$ is transferred along with the query bundle and used as $v_{\text{start}}$ on the next node in order to facilitate pruning. Next, $v$ is updated with a new value only when a new accumulator has been inserted into the candidate heap and $v <$ minHeap.min $+ r$. Finally, prior to a transfer the accumulator IDs are gap-coded and compressed with NewPFor and the partial scores are converted from double to single precision.

## 4 Intra-query parallel processing

In this section we introduce our new query processing approach, divided in two parts. In the first part we address intra-query parallelism on different nodes, and in the second part - on the same node. The experimental results and comparison to the baseline approach follow in the next section.

### 4.1 Query parallelism on different nodes

**Fragment pipeline.** In order to overlap the execution of the same query on two consecutive nodes, we divide the document ID range into several sub-ranges, fragments. For a query $q$ we define a fragment size $F_q$, which splits the ID range $[0, |D|)$ into $N_q = \lceil \frac{|D|}{F_q} \rceil$ sub-ranges or fragments. Fragment $i$ covers document IDs $[iF_q, (i+1)F_q)$.

As we illustrate in Fig. 2(a), each sub-query can now be divided into several tasks, each processing the sub-query over a single fragment. For simplicity, we explain the execution of a single query. With the state-of-the-art approach each sub-query is processed as a single task, which includes three steps: (a) decompression of the incoming accumulator set, (b) processing/merging of the posting and accumulator data, and (c) elimination of false-positives and compression of the new accumulator set or extraction of final results. With a fragment-pipeline, all three steps are scaled down to a single fragment. For example, the node processing the first sub-query post-processes, compresses and transfers its partial accumulator set as soon as the first fragment is finished. Then it starts straight on the second fragment. The next node in the route starts processing as soon as the accumulator set corresponding to the first fragment has arrived.

As an alternative to this method we could process each sub-query until the number of non-pruned accumulators would be above a minimum number, then transfer these and resume processing. However, this could lead to one-to-many and many-to-one correspondences between processing tasks on different nodes, and therefore require a complex implementation with many special cases. Our solution simplifies the implementation, as each node has only one-to-one correspondence between incoming and outgoing fragments. Additionally, we avoid delaying the accumulator transfer in order to wait for the next incoming fragment.

So far we look at the processing model where all tasks corresponding to a single sub-query are done by a single thread, the executor. In order to be efficient, these tasks have to reuse the candidate heap, the pruning threshold and the state of the posting list iterators, called sub-query state. The state contains

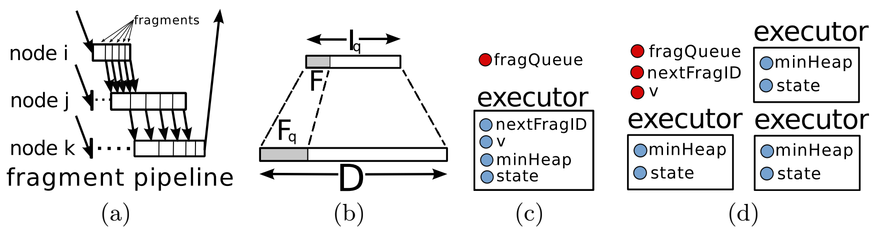


**Fig. 2.** (a) Fragment pipelined query processing. (b) Mapping between $F$ and $F_q$. (c)-(d) Data structures used by (c) non-concurrent and (d) concurrent fragment processing.

information on the number of non-finished iterators, including the current position within the posting list, recently decompressed and fetched data (which can be reused by future tasks), and which of the posting lists can be processed in the skip-mode. Further, as posting list iterators support only next() and skipTo($d$) operations, fragments have to be processed in-order. As Fig. 2(c) shows, a priority queue and a counter are associated with each sub-query to enforce the order. As fragments arrive, they are inserted into the priority queue and the corresponding executor is notified. If the next fragment in the priority queue has the sequence ID corresponding to the counter value, the fragment is processed by the executor and forwarded to the next node, and the counter is increased. If not, the executor suspends processing in order to wait for more data.

Additionally, as the pruning threshold for each sub-query increases gradually, at some point the current pruning threshold of a sub-query $i$, $v_i$, can exceed the current threshold value in the next sub-query $i+1$. Therefore, $v'_i$ seen right after finishing a fragment is packed and transferred along with the accumulators. On the next node, it replaces the current threshold $v_{i+1}$ if $v'_i > v_{i+1}$.

**Fragment size estimation.** As different queries have different processing cost, those more expensive queries are desired to consist of a larger number of fragments than the shorter ones. We assume that the total processing cost of each query is proportional to the total number of candidates produced by a full non-pruned disjunction of query terms, $I_q = \bigcup_{t \in q} I_t$. Next, we introduce a system-dependent (smallest) fragment size $F$, which corresponds to the fragment size used by some hypothetical query that has to consider all of the documents in the collection as potential accumulators. $F$ has to be chosen dependent on the systems settings. In practice it can be tuned during the warm-up or in the run-time. For a particular query $q$, the fragment size $F_q$ can be chosen so that $|I_q|/F = |D|/F_q$ holds. As Fig. 2(b) shows, this equality reflects the correspondence between the document ID space $D$ and the results set $I_q$.

Assuming non-correlated terms, $|I_q|$ can be approximated by Eq. (1). The equation uses the probability that a document does not contain a given term $t$, $(1 - \frac{|I_t|}{|D|})$, to find the probability that a document contains at least one of the query terms, and finally multiplies it by the total number of indexed documents $D$. Then, $F_q$ can be calculated with Eq. (2). As $F_q$ cannot be smaller than $F$ or larger than $|D|$, we further apply Eq. (3).

$$|I_q| \approx |D| \cdot (1 - \prod_{t \in q} (1 - \frac{|I_t|}{|D|})) \tag{1}$$

$$F'_q = \frac{|D|}{|I_q|} \cdot F \approx F / (1 - \prod_{t \in q} (1 - \frac{|I_t|}{|D|})) \tag{2}$$

$$F_q = \begin{cases} F & \text{if } F'_q < F \\ |D| & \text{if } F'_q > |D| \\ F'_q & \text{otherwise} \end{cases} \tag{3}$$

### 4.2 Sub-query parallelism on a single node

**Concurrent fragment processing.** At lower query rates processing nodes cannot fully utilize all of their resources, therefore it can be useful to process the tasks corresponding to the same query concurrently. Processing each fragment completely independent from the others would require a separate sub-query state, candidate heap and pruning threshold, and therefore significantly degrade the performance. Instead, we suggest to use a small number of executors associated with each sub-query and distribute the tasks between them.

When a query $q$ is first received by the node $i$, it initiates $T_{q,i}$ task executors. Each executor initiates its own sub-query state. Similar to the previous description, a priority queue is used to order incoming fragments. In order to ensure that each executor processes fragments by increasing sequence ID and no fragments are left behind, the priority queue and fragment sequence counter are shared between the executors. We illustrate this in Fig. 2(d).

The executors associated with the same query may share the pruning threshold variable and/or the candidate heap. Apart from the experiments presented in the next section, we have evaluated no-share policy against threshold-only and heap-and-threshold. No-share results in a lower performance as pruning efficiency goes down. With a shared candidate heap, synchronized inserts into the heap slow down processing. Our method of choice, threshold-only, is relatively cheap, since $v$ can be marked as volatile and updated by a synchronized setIfGreater($v'$) only when $v' > v$.

Since the candidate heaps are not shared, for the last sub-query, they have to be combined in order to extract the top-$k$ results. This is done by processing the sub-query as long as there are more fragments. The first executor to finish is then chosen as a heap-merger and a heap-queue is associated with the query. Each of the remaining executors, prior to finishing, inserts its candidate heap into the queue. Heaps are then taken by the merger-executor and combined with its own candidate heap. When all of the executor heaps are merged, the final $k$ results are sorted and sent back to the broker.

**Estimation of executor number.** $T_{q,i}$ can be calculated using Eq. (4). First, we introduce a system defined maximum number of executors per query $T_{\max}$ and divide it by the number of queries currently running on this node $Q_{\text{now},i}$ plus one. Additionally, if $N_q$ is too small, the corresponding number of executors should also be smaller. Therefore, we introduce a tunable minimum number of fragments per task, $N_{\text{minpt}}$. The number of executors assigned to a query is therefore the smallest of the two estimates, but not smaller than 1.

$$T_{q,i} = \max(\min(\lfloor \frac{T_{\max}}{Q_{\text{now},i} + 1} \rfloor, \lfloor \frac{N_q}{N_{\text{minpt}}} \rfloor), 1) \tag{4}$$

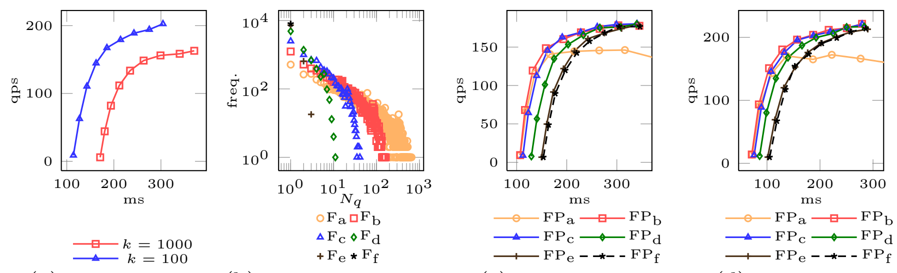
**Multi-stage fragment processing.** As mentioned previously, each task consists of three stages (decompression, processing and compression), but the dependency in fragment execution lies only in the second stage. As an architecture design choice we decode and decompress incoming packages by small executor tasks outside of the query processing executors. Therefore, as incoming fragments enter the priority queue, they are already decompressed and ready for execution. This simplifies the implementation of the methods and reduces the amount of work done by the query executor. Separate post-processing and compression of outgoing data could be done by an additional executor or as a number of small executor tasks. However, the improvement is achieved only when there are many idle CPU cores. Otherwise, it only increases the overhead. For this reason, the results presented in the next section exclude separate fragment post-processing. Finally, we separate the query itself from accumulator transfer. Instead of sending a query bundle to the first node in the route, the broker multi-casts it to all of the query nodes. Therefore, each node can parse the query itself, access the lexicon and initiate the query executor(s) prior to receiving any accumulator data.

## 5 Experiments

In this section we evaluate the performance of three query processing methods: the baseline method (P) described in Section 3, the non-concurrent fragment pipeline (FP) described in Section 4.1 and concurrent fragment pipeline (CFP) described in Section 4.2. Further, both FP and CFP apply the ideas described in the multi-stage fragment processing part of the last section.

For the experiments we use the 426GB TREC GOV2 corpus. With both stemming and stop-word removal applied, the 9.4GB distributed index contains 15.4 mil. unique terms, 25.2 mil. documents, 4.7 bil. pointers and 16.3 bil. tokens. For query processing we use the Okapi BM-25 model. The performance is evaluated using the first 10 000 queries from the Terabyte Track 05 Efficiency Topics that match at least one indexed term, where the first 2 000 queries are used for warm-up and the next 8 000 (evaluation set) to measure the performance. Among the 8 000 test queries, the average query has 2.9 terms and 2.44 sub-queries. For the experiments we use a 9 node cluster interconnected with a Gigabit network. Each node has two 2.0GHz Quad-Core CPUs, 8GB memory and a SATA disk. Our framework is implemented in Java and uses Java NIO and Netty 3.2.3 for fast disk access and network transfer. For disk access we use 16KB blocks. Finally, we use the default Linux disk-cache policy, but drop the cache before each run. Every experiment is repeated twice and the average value is reported.

**Baseline.** Fig. 3(a) illustrates the average throughput and query latency of the baseline method. Marks on each plot correspond to query multiprogramming



(a) Baseline (P) (b) Distribution of $N_q$ (c) FP, $k = 1000$ (d) FP, $k = 100$

**Fig. 3.** (a) Baseline performance. (b)-(c) Distribution of $N_q$ values in the evaluation set. (c)-(d) Performance of FP. In (b)-(d), different plots correspond to different values of $F$: a - 32 768, b - 131 072, c - 524 288, d - 2 097 152, e - 8 388 608, f - 33 554 432.

levels {1, 8, 16, 24, 32, 40, 56, 64}, which are the maximum number of queries concurrently executing in the cluster. We stop at 64 concurrent queries (cq), which corresponds to the total number of processor cores held by the processing nodes. As the figure shows, for $k = 1000$ (i.e. when the result set is restricted to top-1000) the shortest average latency (1cq) is about 170ms, which corresponds to 5.85 queries per second (qps). As we increase the query load, both throughput and latency increase. Over the time, due to to a limited amount of resources and increasing load, the increase in latency dominates over the increase in throughput. The highest throughput is reached at 64cq, 162qps corresponding to 370ms. $k = 100$ has similar results. However, as the $k$-th best candidate score is used to calculate $v$, smaller $k$ decreases the amount of data to be read, processed and transferred, and therefore improves the performance. For $k = 100$, the shortest average latency is about 114ms (8.75qps) and the highest throughput is 203qps, reached at 304ms (64cq).

**Fragment pipeline.** Fig. 3(b) illustrates the distribution of $N_q$ values in the evaluation set. Different plots correspond to different values of $F$. Here we use $F_a = 128^2 \times 2 = 32768$ as a baseline and increment the value by 4 to calculated $F_{b..f}$. As the figure shows, with $F_a$, the $N_q$ lies within [1, 654]. With $F_b$, $F_c$ and $F_d$ the ranges correspond to [1, 164], [1, 41] and [1, 11]. The of frequency of values changes also towards the smaller values. With $F_a$, $N_q = 1$ occurs 514 times. This corresponds to 1243 times with $F_b$, 2499 times with $F_c$, 4871 with $F_d$. With $F_e$, the only value of $N_q$ is 1.

Fig. 3(c)-3(d) demonstrate the performance of FP. Both figures show that the shortest query latency decreases with $F$. However, with $F_a$ the method reaches a starvation point at 24cq, caused by a too large number of network messages and a significantly large processing overhead due to fragmentation. Further, the results show that the difference between $F_b$ and $F_c$, and between $F_e$ and $F_f$ is less significant. At the same time, e.g., $F_b$ has a slightly shorter latency at 1cq than $F_c$, but it reaches a slightly smaller maximum throughput at 64cq. For $k = 1000$ $F_b$ reaches the maximum throughput at 56cq. At 64cq it is outperformed by the other methods. These results show that FP can significantly improve the performance at lower multiprogramming levels. For higher levels (64cq and above), the overhead from fragmentation degrades the performance.

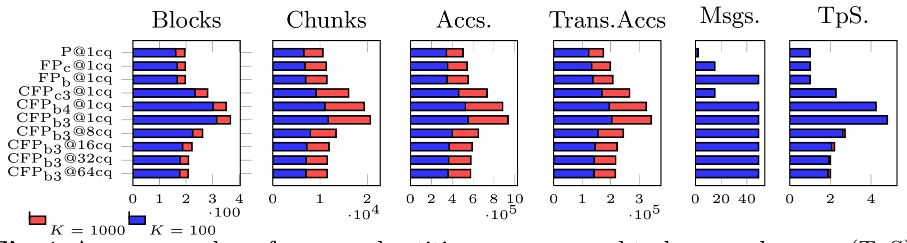
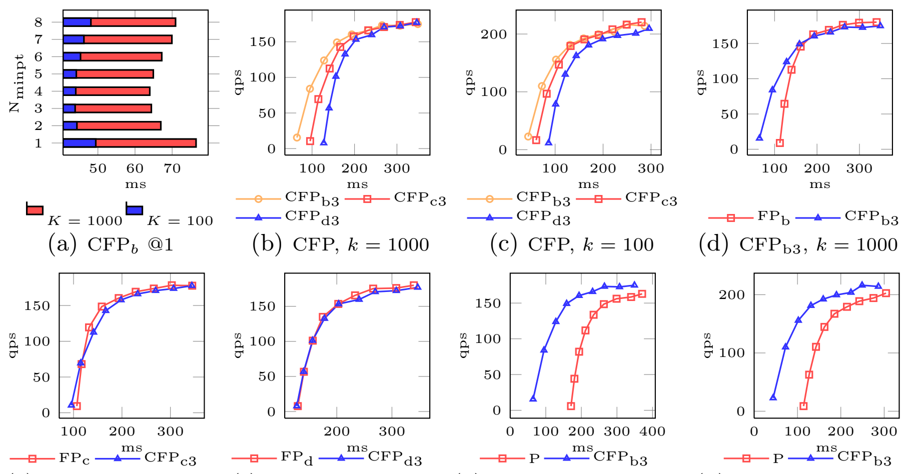


**Fig. 4.** Average number of processed entities per query and tasks per sub-query (TpS) observed at the given multiprogramming level.



(a) CFP$_b$ @1 (b) CFP, $k = 1000$ (c) CFP, $k = 100$ (d) CFP$_{b3}$, $k = 1000$

(e) CFP$_{c3}$, $k = 1000$ (f) CFP$_{d3}$, $k = 1000$ (g) CFP$_{b3}$/P, $k = 1000$ (h) CFP$_{b3}$/P, $k = 100$

**Fig. 5.** Throughput and latency with the concurrent fragment pipeline (CFP).

Fig. 4 shows the average number of read data blocks, decompressed chunks, created and transferred accumulators and sent messages for queries in the evaluation set. As pruning efficiency degrades with smaller fragments, the figure shows a small increase in the measured numbers for blocks, chunks and accumulators for FP with $F_b$ (FP$_b$) and $F_c$ (FP$_c$). At the same time, the number of network messages per query (except the result message) increases from 2.44 with P to 49 with FP$_c$.

**Concurrent fragment pipeline.** Fig. 5(a) illustrates the performance of CFP with one query at a time (1cq), $F_b$ and varied $N_{\text{minpt}}$. We use $T_{\max} = 8$, since each node has 8 CPU cores. For both $k = 100$ and $k = 1000$, the shortest latency is observed at $N_{\text{minpt}} = 2$ or 3. For $N_{\text{minpt}} = 1$ the improvement is limited due to decreased pruning efficiency and increased synchronization overhead. Fig. 4 shows that the amount of read, decompressed and transferred data increases with smaller $F$. In the figure, $N_{\text{minpt}}$ is indicated by the second subscript. At the same time, the amount of work that can be done concurrently increases along with $N_{\text{minpt}}$. For $N_{\text{minpt}} = 2$ and 3, the trade-off ratio between parallelism and overhead is therefore optimal. Similar results were observed for $F_c$ and $F_d$.

Fig. 5(b)-5(c) demonstrate the performance of CFP with varied fragment size. At low multiprogramming levels the difference between $F_b$ and $F_c$ is quite significant and it is clear that smaller fragment sizes speed-up query processing. However, Fig. 4 shows a significant increase in the processed data (1cq), which decreases with increasing multiprogramming. Despite this, at higher levels intra-query concurrency starts to degrade the performance. Fig. 5(d)-5(f) show a comparison between CFP and the corresponding FP runs. The figure shows that the maximum throughput decreases for both $F_a$, $F_b$ and $F_c$, however while $\mathrm{CFP_{b3}}$ significantly reduces the latency compared to $\mathrm{FP_3}$, $\mathrm{CFP_{d3}}$ only degrades the performance of $\mathrm{FP_d}$. The last observation can be explained by a small number of fragments per-query ($F_\mathrm{d}$ gives $N_q \in [1, 11]$) and a relative high overhead cost.

The final comparison between $\mathrm{CFP_{b3}}$ and the baseline method is illustrated in Fig. 5(g)-5(h). As the results show, at 1cq the query latency is 43.9ms for $k = 100$ and 64.4ms for $k = 1000$. This corresponds to a latency decrease by 2.6 times. At the same time, the maximum throughput has increased by 6.6% ($\mathrm{CFP_{b3}}$, $k = 100$, 56cq, 216.2 qps at 249.0ms) to 7.4% ($k = 1000$, 64cq, 174.9qps at 349.3ms). The most important, CFP allows to reach the same throughput as the baseline method at a half of the latency and using a lower multiprogramming level, which satisfies our main objective. For example, for $k = 100$ we reach 190qps at 159ms (32cq) compared to 275ms (56cq) with the baseline method. At higher levels ($\gg$64cq) CFP might be outperformed by P. Therefore, in order to optimize the total performance, a practical search engine could switch between P, FP and CFP depending on the query load.

## 6 Conclusion and further work

In this work we have presented an efficient extension of the pipelined query processing that exploits intra-query concurrency and significantly improves query latency. Our results indicate more than 2.6 times latency improvement on the single-query case. For a general case, we are able to achieve similar throughput with almost half of the latency. Further work can be done in several directions. First, due to the assignment strategy, posting lists stored at the first few nodes are relatively short. Therefore, a hybrid combination between pipelined and non-pipelined execution, where the first nodes perform only fetching and transfer of the posting data to a node later in the query route, can significantly improve the performance. However, this technique requires careful load-balancing. A dynamic load balancing strategy can be explored as the second direction of the future work. Third, the index size and the number of nodes used in our experiments are relatively small. In the future, we plan to evaluate our method on a larger document collection and/or a larger cluster. Finally, we can also think of an efficient combination of the pipelined query execution with impact-ordered lists and bulk-synchronous processing similar to the methods presented by Marin *et al.* [9, 10].

**Acknowledgements.** This work was supported by the Information Access Disruptions Centre (http://iad-centre.no) funded by the Research Council of Norway and the Norwegian University of Science and Technology. The authors thank João B. Rocha-Junior for the useful advices and comments on the paper.

## References

1. C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, 2001.
2. S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010.
3. S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *SIGIR*, 2011.
4. E. Feuerstein, M. Marin, M. Mizrahi, V. Gil-Costa, and R. Baeza-Yates. Two-dimensional distributed inverted files. In *SPIRE*, 2009.
5. S. Jonassen and S. E. Bratsberg. A combined semi-pipelined query processing architecture for distributed full-text retrieval. In *WISE*, 2010.
6. S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *ECIR*, 2011.
7. N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, 2005.
8. C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *InfoScale*, 2007.
9. M. Marin and V. Gil-Costa. High-performance distributed inverted files. In *CIKM*, 2007.
10. M. Marin, V. Gil-Costa, C. Bonacic, R. Baeza-Yates, and I. Scherson. Sync/async parallel search for the efficient design and construction of web search engines. *Parallel Computing*, 2010.
11. A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, 2006.
12. A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 2007.
13. B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *DL*, 1998.
14. T. Strohman and W. Croft. Efficient document retrieval in main memory. In *SIGIR*, 2007.
15. A. Tomasic and H. Garcia-Molina. Query processing and inverted indices in shared nothing text document information retrieval systems. *The VLDB Journal*, 1993.
16. H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 1995.
17. W. Webber. Design and evaluation of a pipelined distributed information retrieval architecture. Master's thesis, 2007.
18. W. Xi, O. Sornil, M. Luo, and E. Fox. Hybrid partition inverted files: Experimental validation. In *ECDL*, 2002.
19. H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, 2009.
20. J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. *Paral. and Dist. Proc. Symp., Int.*, 2007.