Serverless BM25 Search and BERT Reranking

Mayank Anand, Jiarui Zhang, Shane Ding, Ji Xin and Jimmy Lin

University of Waterloo, Canada

Abstract

The retrieve–rerank pipeline is a well-established architecture for search applications, typically with first-stage retrieval using keyword search followed by reranking with a transformer-based model. In deploying such an architecture in the cloud, developers must devote considerable effort to resource provisioning and management: typically, the goal is to optimize the infrastructure configuration (number and type of server instance) to achieve certain performance characteristics (latency, throughput, etc.) while reducing operating costs. In this paper, we introduce a serverless prototype of the retrieve–rerank pipeline for search using Amazon Web Services (AWS), comprised of BM25 for first-stage retrieval using Lucene followed by reranking with the monoBERT model using Hugging Face Transformers. The advantage of a serverless design is that a cloud provider shoulders the burden of operational management, for example, allocating server instances and scaling with query load. We experimentally show with the popular MS MARCO passage ranking test collection that compared to a traditional server-based deployment, our serverless implementation (1) retains the same level of effectiveness, (2) can reduce average latency by exploiting massive parallelism, and (3) incurs comparable costs if the service is expected to be idle for some fraction of the time. Our implementation is open-sourced at https://github.com/castorini/serverless-bert-reranking.

Keywords

multi-stage ranking architectures, transformers, monoBERT

1. Introduction

It is a common practice today for search engines to adopt a retrieve–rerank architecture, for example, with keyword search as first-stage retrieval followed by a transformer-based model for reranking [1]. This represents a simple instantiation of a multi-stage retrieval architecture [2] that is widely used in production at scale [3, 4, 5, 6]. In terms of deployments, individual servers (today, typically virtualized instances in the cloud) form the basic building blocks for search applications. Persistent services running on a cluster cooperate to provide the various functionalities that comprise the complete application. To scale out, the standard practice is to adopt a replicated, document-partitioned architecture [7, 8, 9].

This design has two important implications: First, the services must exist as always-on, long-running processes, ready to handle requests at any moment. This presents a floor on resource consumption, as costs are incurred even when the service is idle. Second, scaling up and down in response to query load must be performed at the granularity of servers, usually through replication and load balancing. Thus, a server-based design means that when the query load is low, even a single server may be over-provisioning; for robust failover, a minimal installation typically runs two servers, additionally contributing to idle (wasted) resources. As the query load increases, to maintain the same level of performance, more server instances need to be provisioned. As query load decreases, these extra instances must then be destroyed. To cope with variable load robustly, developers need to build logic to dynamically spin up and down instances, which may be complex and error prone. While these are solvable engineering challenges, we wonder if there's a better way. It would be desirable if we could scale up and down seamlessly, without effort—and ideally, all the way down to *zero*. That is, if there are no incoming queries, can we not have to pay anything?

Serverless architectures to the rescue! In this paper, we present a serverless prototype of the retrieve–rerank pipeline for search using Amazon Web Services (AWS), comprised of BM25 for first-stage retrieval using Lucene followed by reranking with the monoBERT model using Hugging Face Transformers. We describe our design and present experimental results with the MS MARCO passage ranking test collection. In addition to the ability to completely offload operational management, we believe that there are two scenarios where our serverless design is particularly compelling: (1) a search application that handles low query volumes, where server instances may be idle most of the time, and (2) a search application where incoming requests may be bursty, for example, a service endpoint that is periodically invoked by some other component. While our prototype exhibits a number of limitations at present, it perhaps offers a blueprint for a very different approach to how future search applications may be built.

DESIRES 2021 – 2nd International Conference on Design of Experimental Search & Information REtrieval Systems, September 15–18, 2021, Padua, Italy

mayank.anand@uwaterloo.ca (M. Anand); jiarui.zhang@uwaterloo.ca (J. Zhang); s44ding@uwaterloo.ca (S. Ding); ji.xin@uwaterloo.ca (J. Xin); jimmylin@uwaterloo.ca (J. Lin)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-2950/paper-01.pdf

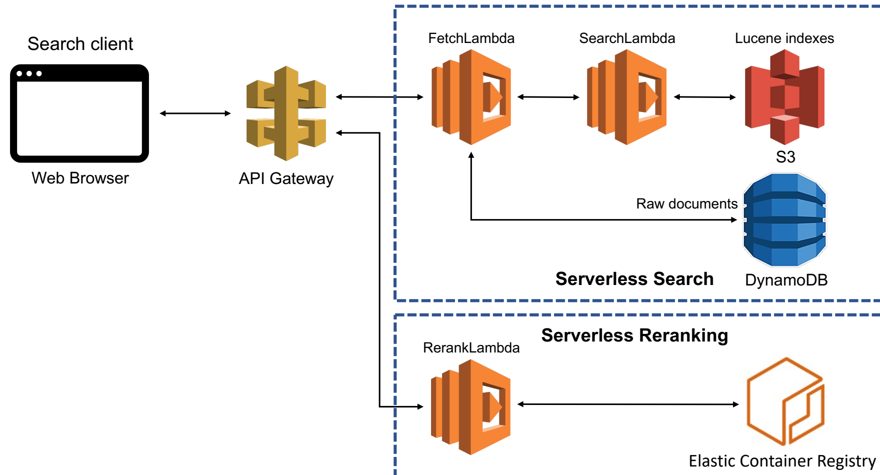

Figure 1: The architecture of our serverless search and reranking prototype.

2. Background and Related Work

The development of cloud technologies can be characterized as continuing disaggregation of computing components. In the early days, the cloud meant dynamic, readily-available, easy-to-provision virtual machines. Today, however, there exists a myriad of services, offered by all the major cloud providers, that deliver computing capabilities in a much more fine-grained manner under a pay-as-you-go model.

One particularly interesting development is the rise of so-called Function-as-a-Service (FaaS) offerings: The developer provides a block of code with well-known entry and exit points, and the cloud provider handles all other aspects of execution—provisioning resources to execute those functions, scaling up and down to match a particular load, etc., all under a per-invocation cost model. Combined with storage- and database-as-a-service offerings, it is now possible to write end-to-end serverless applications where the abstraction of a server is completely absent. To be clear, a serverless design does not mean that we can somehow compute without servers. Rather, it means that the developer no longer needs to explicitly manage server instances. Instead, the cloud provider shoulders the burden of operational management, thus freeing the developer to focus on implementing the application logic.

Researchers have explored serverless architectures for a variety of applications [10, 11, 12, 13, 14], but most relevant to this work is serverless search: Crane and Lin [15] previously demonstrated a working prototype on Amazon Web Services. In their design, postings lists are stored in the DynamoDB data store and query execution is handled by Lambda (Amazon's FaaS offering). Their work demonstrated the feasibility of serverless search, but has a number of shortcomings. In particular, their prototype required custom code, which presents barriers to broad adoption. Lin [16] addressed this shortcoming by demonstrating how the open-source Lucene search library can be packaged in a serverless design with minimal custom code to achieve query latencies capable of supporting interactive retrieval. This serverless Lucene prototype forms the starting point of our work, where we further add serverless reranking with transformer-based models to demonstrate a full serverless retrieve–rerank pipeline.

3. Serverless Architecture

We present and evaluate a working implementation of a serverless retrieve–rerank pipeline with the architecture shown in Figure 1. In this work, Amazon Web Services (AWS) was selected as the cloud platform: in particular, Lambda, the AWS Function-as-a-Service offering, provides the core building block in our design. Nevertheless, other popular cloud providers offer comparable services that can provide alternative implementations.

Our design is comprised of two distinct components, serverless search and serverless reranking, described below. The serverless search component is built on the serverless Lucene prototype presented by Lin [16], while the serverless reranking component has not been described anywhere else.

3.1. Serverless Search

An important desideratum of our work is to build serverless search on the open-source Lucene search library,

which has emerged as the *de facto* platform for developing real-world search applications, typically via OpenSearch, Elasticsearch, Solr, or other components in the broader ecosystem. Other than a few commercial search engine companies that deploy custom infrastructure (for which the serverless design would not be of interest anyway), Lucene dominates the search landscape, with deployments at organizations ranging from Bloomberg to Twitter to Wikipedia. Furthermore, the use of Lucene for academic research has been gaining traction [17, 18, 19]. Thus, to increase the potential for broader impact and adoption, our focus is to leverage as much of the existing Lucene codebase as possible.

The design of serverless architectures hinges around the decoupling of state from stateless code. In the context of search, "state" is captured by the inverted index and other related data structures, while query evaluation (i.e., postings traversal) can be considered stateless. Thus, it is only natural to package Lucene's query evaluation code (IndexReader, IndexSearcher, etc.) into a Lambda function—the SearchLambda in Figure 1. The index structures (assumed to have been generated elsewhere) can be stored in S3, Amazon's persistent object store.

How do we "connect" Lucene code (running in a Lambda) with index structures stored in S3? Fortunately, Lucene's Directory interface provides a low-level abstraction for reading index structures (at the level of reading bytes from streams, seeking to different byte offset positions, etc.). Thus, it suffices to provide a custom Directory implementation built with Amazon's S3 API, and then use this implementation for reading the indexes. Critically, all other parts of the Lucene query evaluation stack remain unchanged—instead of consuming bytes from a local drive (for example), the bytes are now streamed across the datacenter network from S3.

Given this design, an important issue of course is the performance of (remote network) reads from S3. This is solved by caching; that is, a custom S3Directory implementation reads data into memory and thus the overall design is no different from main-memory search engines, which are quite commonplace today both in the academic literature [20, 21, 22] as well as in production deployments [9, 23]. In order to understand how this caching mechanism interacts with Lambda execution, it is necessary to understand at a high level how Amazon handles FaaS execution.

Behind the scenes, Amazon is provisioning containers to execute the Lambda; it controls how many containers are running to satisfy a particular load, automatically scales up and down the number of containers, and performs load balancing. Therefore, code execution can either occur on a "warm" instance (i.e., already running container) or a "cold" instance. For a "warm" instance, query evaluation proceeds without overhead as the index structures have already been loaded into memory; initial execution on a "cold" instance, however, carries the additional startup costs associated with populating the cache. This is not unlike any other in-memory system, and Lambda execution incurs no performance penalty in steady state.

To complete the architecture shown in Figure 1, there are a few more components to describe: Raw documents are stored in DynamoDB (organized as a simple key–value store). Another Lambda, the FetchLambda calls the SearchLambda to generate a ranking, and then issues concurrent, batched calls to DynamoDB to retrieve the actual document text (which is needed for reranking). The FetchLambda can be triggered through a REST endpoint provided by the API Gateway. The final product is a service that takes a query and returns a list of documents (complete with their contents) accessible to a search client (e.g., in a web browser).

Although in principle the FetchLambda and the SearchLambda can be combined, we have kept the two separate to support future scale out. A partitioned architecture can be implementation by multiple SearchLambda instances, each responsible for its own index partition, in which case FetchLambda can serve as a central broker.

3.2. Serverless Reranking

In our design, BM25 results from first-stage retrieval are fed to monoBERT, a standard cross-encoder, for reranking. In monoBERT, inference is performed on all candidate documents: an input template comprised of the query and the document text is fed to a fine-tuned BERT model, which produces a relevance score. All candidate documents are then sorted by these scores. Previous studies have shown that this approach is both simple and effective [1]. In this work, we adopt a monoBERT variant called Early Exiting monoBERT [24], which increases the efficiency of the BERT backbone by adding in "early exits" that allow the inference process to terminate early if the model is confident in its decisions. Our implementation is based on Hugging Face Transformers [25] and PyTorch [26].

Conceptually, serverless ranking is straightforward because the operation is stateless and embarrassingly parallel. We simply need to generate a relevance score for each document, which can proceed independently. The obvious implementation is to wrap model inference in a Lambda, and this is exactly what we do with the RerankLambda, as shown in Figure 1. There are, however, two engineering challenges, discussed below.

First, neural inference typically requires GPUs to achieve latencies that are sufficiently low to support interactive applications, but AWS Lambda invocations are limited to CPUs. We mitigate this limitation with the early-exit model optimizations described above as well

as by exploiting the parallelism provided by the FaaS design (more details below).

The second challenge concerns the size of the Lambda deployment package. Presently, AWS places a limit of 250 MB, which is insufficient for both our model and the neural inference stack (Hugging Face Transformers and PyTorch). One straightforward solution is to download the reranker model at execution time, directly from S3 to the temporary directory attached to the Lambda instance. However, this solution is inefficient because the model must be downloaded every time a new execution environment is created. Instead, we directly incorporated our fine-tuned model and the entire execution stack into a container built on the AWS base image for Lambda, where the size limit is 10 GB. We then uploaded the image to ECR, Amazon's fully-managed container registry service, which provides fast and highly-available access. This way, AWS is able to optimize resource provisioning, for example, caching the image closer to where the FaaS invocation occurs. Since the reranking model is already part of the container image, we have eliminated all external dependencies.

As shown in Figure 1, the reranker service endpoint (RerankLambda) is accessible from the API Gateway via HTTP. It receives a JSON request structure comprising the query and (document id, content) pairs to be reranked, performs model inference, and returns a JSON structure with (document id, score) pairs.

In our current prototype, we have completely decoupled serverless search from serverless reranking, but this design imposes some unnecessary data movement: the document contents are returned to the client (across the network) and then sent right back to the RerankLambda for reranking. It would be straightforward to more tightly couple the search and reranking components, but we have currently not done so, primarily because the savings would be modest at best. The additional costs of this extra data transfer are small compared to the costs associated with neural network inference.

4. Experiments and Results

We evaluated our serverless search and reranking prototype using the popular MS MARCO passage ranking test collection [27], which comprises of 8.8M documents (passages). Inverted indexes for the collection were built using the Anserini toolkit [28] and then uploaded to S3. Separately, the raw document texts were inserted into DynamoDB using a custom ingestion script.

In our experiments, we retrieved 1000 hits using BM25 for each query and reranked all of those hits. The SearchLambda returns only the document ids of the retrieval results; since BERT reranking requires the document contents as well, the FetchLambda issues concurrent queries to obtain document contents from DynamoDB in batches of 100 documents.

Table 1
Effectiveness comparisons on the development set of the MS MARCO passage ranking test collection.

Configuration	MRR@10
BM25	0.18
BM25 + Early Exiting monoBERT	0.34

To speed up BERT reranking, we issued parallel RerankLambda requests, each with the query and ten candidate documents. That is, each invocation processed ten documents, and therefore to rerank 1000 hits, we had to issue 100 requests in parallel. With Lambda, this amount of parallelism is easy to obtain—after all, this is exactly the point of FaaS. In principle, we could even issue 1000 parallel requests, each scoring a single document, but we did not try this configuration in our experiments. For the reranking model, Early Exiting monoBERT, we followed the guidance in Xin et al. [24] and selected $\tau_p = 1.0$ and $\tau_n = 0.9$ (third row of Table 1 in the paper). Based on the reported results: with only a 1% drop in MRR, this setting provided $2.9\times$ acceleration, which means that on average, each inference exits the 12-layer transformer model after around four layers.

To evaluate retrieval effectiveness, we ran inference on the entire development set of the MS MARCO passage ranking test collection (6980 queries), using both the serverless prototype and a comparable server-based configuration; MRR@10 scores are shown in Table 1. We encountered minor issues resulting from the encoding of special characters, which translated into very small differences in effectiveness between the two designs (third digit after the decimal point). These issues aside, we can verify that our serverless deployment retains the same level of effectiveness as a server-based design.

To evaluate retrieval latency and cost, we further performed search and reranking on 100 queries from the development set of the MS MARCO passage ranking test collection to obtain more detailed logging data. We measured component latency as well as end-to-end latency from the client side. Table 2 provides a breakdown in terms of mean, 50th, and 99th percentile latency. As we can observe, end-to-end latency is dominated by serverless reranking, due to the computationally intensive nature of neural inference. Before these experiments, we conducted multiple trials to warm up the SearchLambda and RerankLambda instances.

Based on the latency measurements, we estimated operating costs (in US dollars). AWS Lambda charges based on the number of function invocations as well as the duration of the function execution. The pricing also reflects the amount of memory allocated to the function;

Table 2
Component and end-to-end latency and cost based on 100 queries from the development set of the MS MARCO passage ranking test collection. Latency is reported per query, while cost is reported per 100 queries.

Stage	Latency (s/Q)			Cost
	Mean	P50	P99	(/100Q)
BM25	0.65	0.65	0.92	$0.022
DynamoDB Fetch	0.95	0.96	1.06	-
BERT reranking	11.21	10.64	17.90	$15.90
End to end	12.81	12.24	19.35	$16.00
BERT reranking (V100)	26.21	25.52	36.64	$2.20

resources beyond CPU are allocated proportionally based on memory. In our case, we allocated the maximum, 10240 MB. At present, the costs are $0.20 per 1M requests and $0.0000166667 for every GB-second duration; in our case, the per-request charge is negligible. Thus, we estimated compute costs as duration of compute (seconds) $\times$ memory allocated (GB) $\times$ 0.0000166667. For ease of interpretation, we report costs in terms of 100 queries, shown in Table 2. DynamoDB costs are computed according to a complex set of rules that are hard to directly estimate. However, for our experiments, these costs are negligible compared to the other components.

To compare the cost of our serverless prototype with a standard server-based deployment, we also set up our reranking pipeline on a local server with a single NVIDIA V100 GPU. Here, we focus on BERT reranking latency only, as the contributions from the other components are negligible. Based on these latency measurements, we estimated query costs by looking up the per-hour (on-demand) prices of V100 servers from AWS and Azure, both of which provide similar pricing (we used $3.05 per hour as the basis of our calculations). These results are also reported in Table 2.

What do we make of these experimental results? The latency for GPU-based reranking is admittedly longer than comparable figures reported in similar experiments [29]. We attribute this to the lack of batch inference in our implementation. With this caveat in mind, we see that serverless BERT reranking is able to achieve lower latency with only CPUs. No doubt some of this gap is due to our sub-optimal implementation, but the more interesting point is that the serverless design allows us to arbitrarily parallelize Lambda invocations. In our setup, we issued 100 parallel SearchLambda requests, each performing inference on ten documents. To reduce latency further, we could increase parallelism even more, for example, dispatching 1000 parallel requests, each performing inference on a single document. Based on the Lambda pricing model, this should have no appreciable impact on cost. Thus, the lower bound on end-to-end latency is in theory limited by CPU-based inference on a single document.

Nevertheless, it is clear that on a per-query basis, our serverless design is 7–8$\times$ more expensive than a traditional server-based deployment. This is of course expected, and there are two components to this gap. First, the per unit time cost of serverless components must sum up to more than the cost of a comparable server; otherwise, AWS would be losing money on serverless offerings. Second, decomposing a server-based application into a serverless design introduces friction (e.g., unnecessary data movement and network communication). In our specific case, there is the additional difference between GPU vs. CPU neural inference. However, how much each of these factors contributes to the cost difference is difficult to determine.

Summarizing the "bottom line" based on our experimental results: If we expect a server to be idle 85–90% of the time, a serverless deployment is more cost efficient. Beyond costs alone, a serverless design exhibits all the potential advantages we have already discussed: minimal operational burden along with seamless scalability down to zero (zero queries, zero cost) and up to arbitrarily large query loads. Whether these tradeoffs are worthwhile, of course, will depend on the exact operational scenario.

5. Future Work and Conclusions

At a high level, a serverless design provides operational advantages and cost efficiencies for low-load applications, where in traditional server-based designs the developer must still pay for idle servers. In our experiments, this "breakeven point" is around 85–90% idle, but it is important to note that our experimental results reflect a snapshot at a specific point in time, based on a specific implementation. Below, we discuss some of the factors that may play a role in this calculus, and how they might change over time.

First, there is a general downward trend in AWS costs over time as computing capabilities advance. However, the relative costs between storage, server instances, and FaaS invocation may not be stable. These differences will impact costs over time, but unfortunately, the developer has little control over pricing.

Second, costs will be affected by different architecture and implementation choices, including future innovations. The reader may have noticed that in our experiments, end-to-end latency for both the serverless and server-based deployments are still outside the range of what is acceptable for an interactive application. There are various options to reduce latency: we can choose to rerank fewer BM25 results, in which case we are trading off effectiveness for efficiency. As we have already mentioned, Lambda could support greater parallelism

(thus lower latency) without increasing costs. For the server-based design, we can also rerank in parallel, but this would increase costs (e.g., requiring a larger server with more GPUs). These considerations seem to be in favor of the serverless design with its per-invocation cost model and seamless scalability.

Neural inference forms the biggest component of both latency and cost, and there is much research on models that support faster and more efficient inference. Examples include ALBERT [30], TinyBERT [31], and QBERT [32], all of which can serve as drop-in replacements for our current reranker. These improvements will benefit both the serverless and server-based design, and to a large extent we can ride the wave of future innovations in NLP "for free". The interesting question, however, is whether some of these innovations will differentially impact CPU-based vs. GPU-based inference, or perhaps in the future FaaS offerings might support GPUs. We do not have answers at present, but future explorations of these issues would be interesting.

To conclude, in this work we built on an existing serverless Lucene prototype to demonstrate a complete retrieve–rerank search architecture with a transformer-based model. Our experiments allow us to characterize the tradeoffs between serverless and sever-based designs. No doubt the costs of both approaches will change over time as the economics of cloud computing evolve and as technical innovations lead to efficiency improvements. However, the operational advantages of the serverless design will remain. Whether such an architecture will gain widespread adoption remains to be seen, but at the very least this design challenges how we think about the architecture of search applications.

Acknowledgments

This research was supported in part by the Canada First Research Excellence Fund and the Natural Sciences and Engineering Research Council (NSERC) of Canada; computational resources were provided by Compute Ontario and Compute Canada.

References

[1] R. Nogueira, K. Cho, Passage re-ranking with BERT, arXiv:1901.04085 (2019).

[2] J. Lin, R. Nogueira, A. Yates, Pretrained transformers for text ranking: BERT and beyond, arXiv:2010.06467 (2020).

[3] J. Pedersen, Query understanding at Bing, in: Industry Track Keynote at the 33rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2010), Geneva, Switzerland, 2010.

[4] S. Liu, F. Xiao, W. Ou, L. Si, Cascade ranking for operational e-commerce search, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD 2017), Halifax, Nova Scotia, Canada, 2017, pp. 1557–1565.

[5] J.-T. Huang, A. Sharma, S. Sun, L. Xia, D. Zhang, P. Pronin, J. Padmanabhan, G. Ottaviano, L. Yang, Embedding-based retrieval in Facebook search, in: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD 2020), 2020, pp. 2553–2561.

[6] L. Zou, S. Zhang, H. Cai, D. Ma, S. Cheng, D. Shi, Z. Zhu, W. Su, S. Wang, Z. Cheng, D. Yin, Pre-trained language model based ranking in Baidu search, arXiv:2105.11108 (2021).

[7] L. A. Barroso, J. Dean, U. Hölzle, Web search for a planet: The Google cluster architecture, IEEE Micro 23 (2003) 22–28.

[8] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, F. Silvestri, Challenges on distributed web retrieval, in: Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE 2007), Istanbul, Turkey, 2007, pp. 6–20.

[9] J. Dean, Challenges in building large-scale information retrieval systems, in: Keynote Presentation at the Second ACM International Conference on Web Search and Data Mining (WSDM 2009), Barcelona, Spain, 2009.

[10] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, Occupy the cloud: Distributed computing for the 99%, in: Proceedings of the 2017 Symposium on Cloud Computing (SoCC 2017), Santa Clara, California, 2017, pp. 445–451.

[11] Y. Kim, J. Lin, Serverless data analytics with Flint, in: Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD 2018), San Francisco, California, 2018, pp. 451–455.

[12] J. Hellerstein, J. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu, Serverless computing: One step forward, two steps back, arXiv:1812.03651 (2018).

[13] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, K. Winstein, From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers, in: Proceedings of the 2019 USENIX Annual Technical Conference, Renton, Washington, 2019, pp. 475–488.

[14] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, A. Tumanov, Cloudburst: Stateful functions-as-a-service, arXiv:2001.04592 (2020).

[15] M. Crane, J. Lin, An exploration of serverless architectures for information retrieval, in: Proceedings of the 3rd ACM International Conference on the

Theory of Information Retrieval (ICTIR 2017), Amsterdam, The Netherlands, 2017, pp. 241–244.
[16] J. Lin, A prototype of serverless Lucene, arXiv:2002.01447 (2020).
[17] L. Azzopardi, M. Crane, H. Fang, G. Ingersoll, J. Lin, Y. Moshfeghi, H. Scells, P. Yang, G. Zuccon, The Lucene for Information Access and Retrieval Research (LIARR) Workshop at SIGIR 2017, in: Proceedings of the 40th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2017), Tokyo, Japan, 2017, pp. 1429–1430.
[18] L. Azzopardi, Y. Moshfeghi, M. Halvey, R. S. Alkhawaldeh, K. Balog, E. Di Buccio, D. Ceccarelli, J. M. Fernández-Luna, C. Hull, J. Mannix, S. Palchowdhury, Lucene4IR: Developing information retrieval evaluation resources using Lucene, SIGIR Forum 50 (2017) 58–75.
[19] P. Yang, H. Fang, J. Lin, Anserini: Reproducible ranking baselines using Lucene, Journal of Data and Information Quality 10 (2018) Article 16.
[20] T. Strohman, W. B. Croft, Efficient document retrieval in main memory, in: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2007), Amsterdam, The Netherlands, 2007, pp. 175–182.
[21] S. Büttcher, C. L. A. Clarke, Index compression is good, especially for random access, in: Proceedings of the Sixteenth International Conference on Information and Knowledge Management (CIKM 2007), Lisbon, Portugal, 2007, pp. 761–770.
[22] J. Lin, A. Trotman, The role of index compression in score-at-a-time query evaluation, Information Retrieval 20 (2017) 199–220.
[23] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, J. Lin, Earlybird: real-time search at Twitter, in: Proceedings of the 28th International Conference on Data Engineering (ICDE 2012), Washington, D.C., 2012, pp. 1360–1369.
[24] J. Xin, R. Nogueira, Y. Yu, J. Lin, Early exiting BERT for efficient document ranking, in: Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing, 2020, pp. 83–88.
[25] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, A. Rush, Transformers: State-of-the-art natural language processing, in: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2020, pp. 38–45.
[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32 (NeurIPS 2019), Vancouver, Canada, 2019, pp. 8024–8035.
[27] P. Bajaj, D. Campos, N. Craswell, L. Deng, J. Gao, X. Liu, R. Majumder, A. McNamara, B. Mitra, T. Nguyen, M. Rosenberg, X. Song, A. Stoica, S. Tiwary, T. Wang, MS MARCO: A human generated MAchine Reading COmprehension dataset, arXiv:1611.09268 (2016).
[28] P. Yang, H. Fang, J. Lin, Anserini: Enabling the use of Lucene for information retrieval research, in: Proceedings of the 40th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2017), Tokyo, Japan, 2017, pp. 1253–1256.
[29] O. Khattab, M. Zaharia, ColBERT: Efficient and effective passage search via contextualized late interaction over BERT, in: Proceedings of the 43rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2020), 2020, pp. 39–48.
[30] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, R. Soricut, ALBERT: A lite BERT for self-supervised learning of language representations, in: International Conference on Learning Representations, 2020.
[31] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, Q. Liu, TinyBERT: Distilling BERT for natural language understanding, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 4163–4174.
[32] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, K. Keutzer, Q-BERT: Hessian based ultra low precision quantization of BERT, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, 2020, pp. 8815–8821.