Read as Needed: Building WiSER, a Flash-Optimized Search Engine

Jun He and Kan Wu, *University of Wisconsin—Madison;* Sudarsun Kannan, *Rutgers University;* Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*

https://www.usenix.org/conference/fast20/presentation/he

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-12-0

Read as Needed: Building WiSER, a Flash-Optimized Search Engine

Jun He, Kan Wu, Sudarsun Kannan[†], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin–Madison
[†]*Department of Computer Science, Rutgers University*

Abstract

We describe WiSER, a clean-slate search engine designed to exploit high-performance SSDs with the philosophy "read as needed". WiSER utilizes many techniques to deliver high throughput and low latency with a relatively small amount of main memory; the techniques include an optimized data layout, a novel two-way cost-aware Bloom filter, adaptive prefetching, and space-time trade-offs. In a system with memory that is significantly smaller than the working set, these techniques increase storage space usage (up to 50%), but reduce read amplification by up to 3x, increase query throughput by up to 2.7x, and reduce latency by 16x when compared to the state-of-the-art Elasticsearch. We believe that the philosophy of "read as needed" can be applied to more applications as the read performance of storage devices keeps improving.

1 Introduction

Modern solid-state storage devices (SSDs) [19, 20] provide much higher throughput and lower latency than traditional persistent storage such as hard disk drives (HDDs). Currently, flash-based SSDs [19, 20] are readily available; in the near future, even higher performance NVRAM-based systems may supplant flash [4], boosting performance even further.

SSDs exhibit vastly different characteristics from HDDs [24, 36]; as we shift from HDDs to SSDs, the software on top of the storage stack must evolve to harvest the high performance of the SSDs. Thus far, optimization for SSDs has taken place within many important pieces of the storage stack. For example, RocksDB [16], Wisckey [38] and other work [23, 34, 42] have made key-value stores more SSD-friendly; FlashGraph [59], Mosaic [43] and other work [48, 49, 60] optimize graphs for SSDs; SFS [45], F2FS [39] and other work [30, 37] have made systems software more SSD-friendly.

In this evolution, an important category of application has been overlooked: full-text search engines. Search engines are widely used in many industrial and scientific settings today, including popular open-source offerings such as Elasticsearch [7] and Solr [3]. As of the time of this writing, Elasticsearch is ranked 7th among all database engines, higher than well-known systems such as Redis, Cassandra, and SQLite [6]. Elasticsearch is used in important applications, such as at Wikipedia and Github to power text (edited contents or source code) search [7, 22]. They are also widely used for data analytics [7]; for example, Uber uses Elasticsearch to generate dynamic prices in real time based on users' locations.

Furthermore, the challenges in search engines are unique, interesting, and different from the ones in key-value stores, graphs, and system software. The key data structure in a search engine is an inverted index, which maps individual terms (i.e., words) to lists of documents that contain the terms. On top of the inverted index, multiple auxiliary data structures (e.g., posting lists) and technologies (e.g., compression) also play important roles to implement an efficient search engine. In addition to compelling data structures, the unique workloads of search engines also provoke new thoughts on SSD-based optimizations. For example, phrase queries (e.g., "United States") stress multiple data structures in the engine and require careful design.

Search engines pose great challenges to storage systems. First, search engines demand low latency as users often interface with them interactively. Second, search engines demand high data throughput because they retrieve information from a large volume of data. Third, search engines demand high scalability because data grows quickly. Due to these challenges, many search engines eschew using secondary storage, putting most/all data directly into memory instead [13, 15].

However, we believe the advent of faster storage suggests a reexamination of modern search engine design. Given that using RAM for large datasets can be cost prohibitive, can one instead rebuild a search engine to better utilize SSDs to achieve the necessary performance goals with main memory that is significantly smaller than the dataset?

We believe the answer is *yes*, if we re-design the system with the principle *read as needed*. Emerging fast storage devices [14, 18, 29, 36, 57] offer high bandwidth; for example, inexpensive (i.e., $0.17/GB) SSDs currently offer 3.5 GB/s read bandwidth [17], and even higher performance can be provided with multiple devices (e.g., RAID). These high-performance storage devices allow applications to read data as needed, which means that main memory can be used as a staging buffer, instead of a cache; thus, large working sets can be kept within secondary storage and less main memory is required. To read as needed, applications must optimize the efficiency of the data stream flowing from the storage device, through the small buffer (memory), to CPU.

In this paper, we present the design, implementation, and evaluation of *WiSER*, a flash-optimized high-I/O search engine that reads data as needed. WiSER reorganizes traditional search data structures to create and improve the read stream, thus exploiting the bandwidth that modern SSDs provide. First, we propose a technique called cross-stage grouping, which produces locality-oriented data layout and significantly reduces read amplification for queries. Second, we propose a novel two-way cost-aware Bloom filter to reduce I/O for phrase queries. Third, we use adaptive prefetch to reduce latency and improve I/O efficiency. Fourth, we enable a space-time trade-off, increasing space utilization on flash for fewer I/Os; for example, we compress documents individually, which consumes more space than compression in groups, but allows us to retrieve documents with less I/O.

We built WiSER[1] from ground up with 11,000 lines of C++ code, for the following reasons. First, an implementation in C++ allows us to interact with the operating system more closely than the state of the art engine, Elasticsearch, which is written in Java; for example, it allows us to readily prefetch data using OS hints. Second, a fresh implementation allows us to reexamine the limitations of current search engines. For example, by comparing with WiSER, we found that the network implementation in Elasticsearch significantly limits its performance and could be improved. Overall, our clean slate implementation produces highly efficient code, which allows us to apply various flash-optimized techniques to achieve high performance. For some (but not all) workloads, WiSER with limited memory performs better than Elasticsearch with memory that holds the entire working set.

We believe that this paper makes the following contributions. First, we propose a design philosophy, read as needed, and follow the philosophy to build a flash-optimized search engine with several novel techniques. For example, we find that plain Bloom filters employed elsewhere [11, 16, 42] surprisingly increase I/O traffic; consequently, we propose two-way cost-aware Bloom filters, which exploit unique properties of search engine data structures. Second, WiSER significantly reduces the need for vast amounts of memory by exploiting high-bandwidth SSDs to achieve high performance at low cost. Third, we have built a highly efficient search engine: thanks to our proposed techniques and efficient implementation, WiSER delivers higher query throughput (up to 2.7x) and lower latencies (up to 16x) than a state-of-the-art search engine (Elasticsearch) in a low-memory system that we use to stress the engine.

The paper is organized as follows. We introduce the background of search engines in Section 2. We propose techniques for building a flash-optimized search engine in Section 3. We evaluate our flash-optimized engine in Section 4, discuss related work in Section 5, and conclude in Section 6.

[1] WiSER is available at http://research.cs.wisc.edu/adsl/Software/wiser/.

ID	Text
1	I thought about naming the engine CHEESE, but I could not explain CHEE.
2	Fried cheese curds, cheddar cheese sale.
3	Tofu, also known as bean curd, may not pair well with cheese.

Table 1: **An Example of Documents** *An indexer parses the documents to build an inverted index; a document store will keep the original text.*

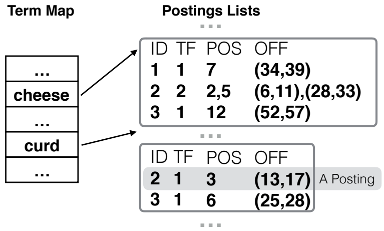


Figure 1: **An Example of Inverted Index** *This figure shows the general contents of inverted index without specific layout information. Term Map allows one to look up the location of the postings list of a term.*

2 Search Engine Background

Search engines play a crucial role in retrieving information from large data collections. Although search engines are designed for text search, they are increasingly being used for data analytics because search engines do not need fixed data schemes and allow flexible data exploration. Popular modern search engines, which share similar features, include Elasticsearch, Solr, and Splunk [6]. Elasticsearch and Solr are open-source projects based on Lucene [1], an information retrieval library. Elasticsearch and Solr wrap Lucene by implementing practical features such as sharding, replication, and network capability. We use Elasticsearch as our baseline as it is the most popular [6] and well-maintained project. Although we only study Elasticsearch, our results also apply to other engines, which share the same underlying library (i.e., Lucene) or key data structures.

2.1 Elasticsearch Data Structures

Search engines allow users to quickly find documents (e.g., text files, web pages) that contain desired information. Documents must be indexed to allow fast searches; the core index structure in popular engines is the *inverted index*, which stores a mapping from a term (e.g., a word) to all the documents that contain the term.

An indexer builds the inverted index. Table 1 shows an example of documents to be indexed. First, the indexer splits a document into tokens by separators such as space and punctuation marks. Second, the indexer transforms the tokens. A common transformation is stemming, which unifies tokens (e.g., curds) to their roots (e.g., curd). The transformed tokens are usually referred to as terms. Finally, the location informa-

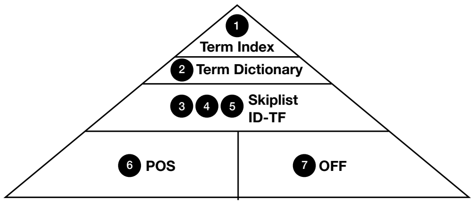


Figure 2: **Inverted Index in Elasticsearch** *Term Index maps a term to an entry in Term Dictionary. A Term Dictionary entry contains metadata about a term (e.g., doc frequency) and multiple pointers pointing to files that contain document IDs and Term Frequencies (ID-TF), positions (POS), and byte offsets (OFF). The number in the figure indicates a typical data access sequence to serve a query. No.3, 4, and 5 indicate the access of skip lists, document ID and Term Frequencies. For Wikipedia, the sizes of each component are Term Index: 4 MB, Term Dictionary: 200 MB, Skiplist.ID.TF: 2.7 GB, POS: 4.8 GB, OFF: 2.8 GB.*

tion of the term is inserted to a list, called a postings list. The resulting inverted index is shown in Figure 1.

A posting contains the location information of a term in a particular document (Figure 1). Such information often includes a document ID, positions, and byte offsets of the term in the document. For example, a position records that term “cheese” is the 2-th and 5-th token in document 2. Positions enable the processing of phrase queries: given a phrase such as “cheese curd”, we know that a document contains the phrase if the first term, “cheese”, is the x-th token and the second term “curd” is the $x+1$-th one. An offset pair records the start and end byte address of a term in the original document. Offsets enable quick highlighting; the engine presents the most relevant parts (with the queried terms highlighted) of a document to the user. A posting also contains information such as term frequency for ranking the corresponding document.

Query processing includes multiple stages: document matching, ranking, phrase matching, highlighting; different types of queries go through different stages. For queries with a single term, an engine executes the following stages: iterating document IDs in a term’s postings list (document matching); calculating the relevance score of each document, which usually uses term frequencies, and finding the top documents (ranking); and highlighting queried terms in the top documents (highlighting). For *AND* queries such as “cheese AND curd”, which look for documents containing multiple terms, document matching includes intersecting the document IDs in multiple postings lists. For the example in Figure 1, intersecting $\{1,2,3\}$ and $\{2,3\}$ produces $\{2,3\}$, which are the IDs of documents that contain both “cheese” and “curd”. For phrase queries, a search engine needs to use positions to perform phrase matching after document matching.

Figure 2 shows the data structures of Elasticsearch. To serve a query with a single term, Elasticsearch follows these steps. First, Elasticsearch locates a postings list by Term Index (1) and a Term Dictionary (2). The Term Index and Term Dictionary contain location information of the skip lists, document IDs, positions, and offsets (details below). Second, the engine will load the skip list, which contains more information for navigating document IDs, term frequencies, positions, and offsets. Third, it will iterate through the document IDs and use the corresponding term frequencies to rank documents. Fourth, after finding the documents with top scores, it will read offsets and document text to generate snippets.

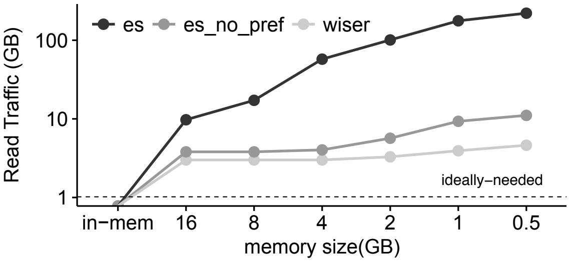


Figure 3: **Read Traffic of Search Engines** *This figure shows read I/O traffic of various search engines as the size of memory decreases.* `es`*: Elasticsearch,* `es_no_pref`*: Elasticsearch without prefetch. Note that serving queries leads to only read traffic. The ideally-needed traffic assumes a byte-addressable storage device.*

2.2 Elasticsearch Performance Problems

Elasticsearch cannot achieve the highest possible performance from the storage system due in part to read amplification. Elasticsearch groups data of different stages into multiple locations and arranges data such that data items in the early stages are smaller. The intention is that data in early stages, which is accessed more frequently than data in later stages, is cached. However, grouping data by stage also could lead to large read amplification.

Figure 3 shows the I/O traffic of a real query workload over Wikipedia; as the amount of memory is decreased, the I/O traffic incurred by Elasticsearch increases significantly. In contrast, the amount of read traffic in WiSER remains relatively low regardless of the amount of memory available.

3 WiSER: A Flash-Optimized Search Engine

Given that SSDs offer high bandwidth, applications that "read data as needed" may be able to run effectively on systems that do not contain enough main memory to cache the entire working set. However, since device bandwidth is not unlimited, applications must carefully control how data is organized and accessed to match the performance characteristics of modern SSD storage [36].

At the highest level, the less I/O an application must perform, the faster that application will complete; since search engine queries involve only read operations, we should *reduce read amplification* as much as possible. Second, retrieving data from SSDs instead of RAM can incur relatively long latency; therefore, applications should *hide I/O latency* as

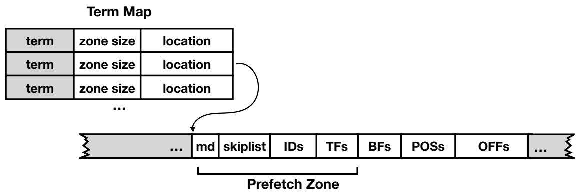


Figure 4: **Structure of WiSER's Inverted Index** *Contents of each postings list are placed together. Within each postings list, IDs, TFs and so on are also placed together to maximize compression ratio, which is the same as Elasticsearch.*

much as possible with prefetching or by overlapping computation with I/O. Third, SSDs deliver higher data bandwidth for larger requests; therefore, an application should organize and structure its data to *enable large requests*.

We introduce techniques that allow WiSER to reduce read amplification, hide I/O latency, and issue large requests. First, *cross-stage data grouping* groups data of different stages and stores it compactly during indexing (reducing read implication and enabling large requests). Second, *two-way cost-aware filtering* employs special Bloom filters to prune paths early and reduce I/O for positions in the inverted index; our Bloom filters are novel in that they are tightly integrated with the query processing pipeline and exploit unique properties of search engines (again reducing read amplification). Third, we *adaptively prefetch* data to reduce query latency; unlike the prefetch employed by Elasticsearch (i.e., OS readahead), our prefetch dynamically adjusts the prefetch size to avoid reading unnecessary data (hiding I/O latency and enabling large requests without increasing read amplification). Fourth, we take advantage of the inexpensive and ample space of SSDs by *trading disk space for I/O speed*; for example, WiSER compresses documents independently and aligns compressed documents to file system blocks to prevent data from crossing multiple blocks (reducing read amplification). We now describe these techniques in more detail.

In discussing our design, we will draw on examples from the English Wikipedia, which is a representative text data set [33, 35, 44, 50, 53, 54, 56]. Its word frequency follows the same distribution (zipf's law) as many other corpuses [41, 50], such as Twitter [47].

3.1 Technique 1: Cross-Stage Data Grouping

One key to building a flash-optimized high-I/O search engine is to reduce read amplification. We propose cross-stage grouping to reduce read amplification for posting lists of small or medium sizes. The processing of such postings lists is critical because most of the postings lists fall into this category. For example, 99% of the postings lists in Wikipedia are small or medium (less than 10,000 documents are in the postings list). Also, search engines often shard large postings lists into such smaller ones to reduce processing latency.

Cross-stage data grouping puts data needed for different stages of a query into continuous and compact blocks on the storage device, which increases block utilization when transferring data for a query. Figure 4 shows the resulting data structures after group; data needed for a query is located in one place and in the order that it will be accessed. Essentially, the grouped data becomes a stream of data, in which each piece of data is expected to be used at most once. Such expectation matches the query processing in a search engine, in which multiple iterators iterate over lists of data. Such streams can flow through a small buffer efficiently with high block utilization and low read amplification.

Grouped data introduces significantly less read amplification than Elasticsearch for small and medium postings lists. Due to highly efficient compression, the space consumed by each postings list is often small; however, due to Elasticsearch's layout, the data required to serve a query is spread across multiple distant locations (Term Dictionary, ID-TF, POS, and OFF) as shown in Figure 2. Elasticsearch's layout increases the I/O traffic and also the number of I/O operations. On the other hand, as shown in Figure 4, the grouped data can often be stored in the one block (e.g., 99% of the terms in Wikipedia can be stored in a block), incurring only one I/O.

3.2 Technique 2: Two-way Cost-aware Filters

Phrase queries are pervasive and are often used to improve search precision [31]. Unfortunately, phrase queries put great pressure on a search engine's storage system, as they require retrieving a large amount of positions data (as described in Section 2). To build a flash-optimized search engine, we must reduce the I/O traffic of phrase queries.

Bloom filters, which can test if an element is a member of a set, are often used to reduce I/O; however, we have found that plain Bloom filters, which are often directly used in data stores [11, 42, 46], increase I/O traffic for phrase queries because *individual* positions data is relatively small due to compression and, therefore, the corresponding Bloom filter can be larger than the positions data.

As a result, we propose special *two-way cost-aware* Bloom filters by exploiting unique properties of search engines to reduce I/O. The design is based on the observation that the postings list sizes of two (or more) terms in a phrase query are often different. Therefore, we construct Bloom filters during indexing to allow us to pick the smaller Bloom filter to filter out a larger amount of positions data during querying. In addition, we design a special bitmap-based structure to store Bloom filters to further reduce I/O. This section gives more details on our Bloom filters.

3.2.1 Problems of plain Bloom filters

A *plain* Bloom filter set contains terms that are after a particular term; for example, the *set.after* of term "cheese" in document 2 of Table 1 contains "curd" and "sale". To test the existence of a phrase "cheese curd", an engine can simply test if "curd" is in *set.after* of "cheese", without reading any positions. If the test result is negative, we stop and consequently avoid reading the corresponding positions. If the test

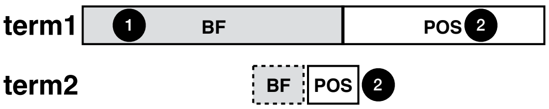


Figure 5: **Phrase Processing With Bloom Filters** *WiSER uses one of the Bloom filters to test the existence of a phrase (1) and then read positions for positive tests to confirm (2).*

result is positive, we must confirm the existence of the phrase by checking positions because false positives are possible in Bloom filters; also, we may have to use positions to locate the phrase within a document for highlighting.

However, we must satisfy the following two conditions to reduce I/O. First, the percentage of negative tests must be high because this is the case where we only read the Bloom filters and avoid other I/O. If the test is positive (a phrase may exist), we have to read both Bloom filters and positions, which increases I/O. Empirically, the percentage of positive results is low for real phrase queries to Wikipedia [21]: only 12% of the tests are positive. Intuitively, the probability for two random terms to form a phrase in a document is also low due to a large number of terms in a regular document. The second condition is that the I/O traffic to the Bloom filters must be smaller than the traffic to positions needed to identify a phrase; otherwise, we would just use the positions.

Meeting the second condition is challenging because the sizes of plain Bloom filters are too large in our case, although they are considered space-efficient in other uses [11, 42]. Bloom filters can be larger than their corresponding positions because positions are already space efficient after compression (delta encoding and bit packing [2]). In addition, Bloom filters are configured to be relatively large because their false positive ratios must be low. The first reason to reduce false positive is to increase negative test results, as mentioned above. The second reason is to avoid reading unnecessary file system blocks. Note that a 4-KB file system block contains positions of hundreds of postings. If any of the positions are requested due to false positive tests, the whole 4-KB block must be read; however, none of the data in the block is useful.

3.2.2 Two-way and cost-aware filtering

We now show how we can reduce I/O traffic to both Bloom filters and positions with cost-aware pruning and a bitmap-based store. To realize it, first we estimate I/O cost and use Bloom filters conditionally (i.e., cost-aware): we only use Bloom filters when the I/O cost of reading Bloom filters is much smaller than the cost of reading positions if Bloom filters are not used. For example, in Figure 5, we will not use Bloom filters for query "term1 term2" as the I/O cost of reading the Bloom filters is too large. We estimate the relative I/O costs of Bloom filter and positions among different terms by term frequencies (available before positions are needed), which is proportional to the sizes of Bloom filters and positions.

Second, we build two Bloom filters for each term to allow filtering in either direction (i.e., two-way): a set for all following terms and another set for all preceding terms of each term. This design is based on the observation that the positions (and other parts) sizes of the terms in a query are often vastly different. With these two Bloom filters, we can apply filters forward or backward, whichever can reduce I/O. For example, in Figure 5, instead of using Bloom filters of term1 to test if term2 is *after* term1, we can now use Bloom filters of term2 to test if term1 is *before* term2. Because the Bloom filters of term2 are much smaller, we can apply it to significantly reduce I/O.

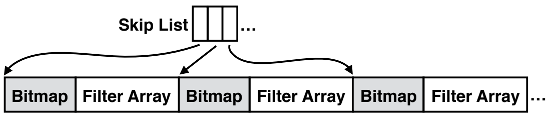


Figure 6: **Bloom Filter Store** *The sizes of the arrays may vary because some Bloom filters contain no entries and thus are not stored in the array.*

To further reduce the size of Bloom filters, we employ a *bitmap-based data layout* to store Bloom filters. Figure 6 shows the data layout. Bloom filters are separated into groups, each of which contains a fixed number of filters (e.g., 128); the groups are indexed by a skip list to allow skipping reading large chunks of filters. In each group, we use a bitmap to mark the empty filters and only store non-empty filters in the array; thus, empty Bloom filters only take one bit of space (in the bitmap). Reducing the space usage of empty filters can significantly reduce overall space usage of Bloom filters because empty filters are common. For instance, about one-third of the filters for Wikipedia are empty. Empty filters of a term come from surrounding punctuation marks and stop words (e.g., "a", "is", "the"), which are not added to filters.

Empirically, we find that expecting five insertions and a false positive probability of 0.0009 in the Bloom filter [12] (each filter is 9-byte) offers a balanced trade-off between space and test accuracy for English Wikipedia; these parameters should be tuned for other data sets. We use the same parameters for all Bloom filters in WiSER because storing parameters would require extra space and steps before testing each filter; one could improve the space overhead and accuracy by limiting the number of parameter sets for the engine and selecting the optimal ones for specific filters from the available sets.

3.3 Technique 3: Adaptive Prefetching

Although the latency of SSDs is low, it is still much higher than that of memory. The relatively high latency of SSDs adds to query processing time, especially the processing of long postings lists which demands a large amount of I/O. If we load one page at a time as needed, query processing will frequently stop and wait for data, which also increases system overhead. In addition, the I/O efficiency will be low due to small request sizes [36].

To mitigate the impact of high I/O latency and improve the I/O efficiency, we propose adaptive prefetching. Prefetching, a commonly used technique, can reduce I/O stall time, increase the size of I/O requests, and reduce the number of requests, which boosts the efficiency of flash devices and reduces system overhead. However, naive prefetching, such as the Linux readahead [10], used by Elasticsearch, suffers from significant read amplification. Linux unconditionally prefetches data of a fixed size (default: 128 KB), which causes high read amplification due to the diverse data sizes needed.

For the best performance, prefetching should adapt to the queries and the structures of persistent data. Among all data in the inverted index, the most commonly accessed data includes metadata, skip lists, document IDs, and term frequencies, which are often accessed together and sequentially; thus we place them together in an area called the *prefetch zone*. Our adaptive approach prefetches data when doing so can bring significant benefits. We prefetch when all prefetch zones involved in a query are larger than a threshold (e.g., 128 KB); we divide the prefetch zone into small prefetch segments to avoid accessing too much data at a time.

To enable adaptive prefetch, WiSER employs prefetch-friendly data structures, as shown in Figure 4. A search engine should know the size of the prefetch zone before reading the posting list (so the prefetch size can be adapted). To enable such prefetching, we hide the size of the prefetch zone in the highest 16 bits of the offset in WiSER's Term Map (the 48 bits left is more than enough to address large files). In addition, the structure in the prefetch zone is also prefetch-friendly. Data in the prefetch zone is placed in the order it is used, which avoid jumping ahead and waiting for data that has not been prefetched. Finally, compressed data is naturally prefetch-friendly. Even if there are data "holes" in the prefetch zone that are unnecessary for some queries, prefetching data with such holes is still beneficial because these holes are usually small due to compression and the improved I/O efficiency can well offset the cost of such small read amplification.

WiSER prefetches by dynamically calling `madvise()` with the `MADV_SEQUENTIAL` hint to readahead in the prefetch zone. We could further improve prefetching with more precise memory management; for example, we could isolate the buffers used for different queries and avoid interference between queries. In addition, Linux prefetches in fixed sizes; we could allow variable sizes to avoid wasting I/O.

3.4 Technique 4: Trade Disk Space for I/O

With a small increase in disk space, WiSER is able to perform less I/O to its document store. We compress each document individually in WiSER, which often increases space usage but avoids reading and decompressing unnecessary documents. Compression algorithms, such as LZ4, achieve better compression when more data is compressed together. As a result, when compressing documents, engines like Elasticsearch put documents into a buffer (default size: 16 KB) and compresses all data in the buffer together. Unfortunately, decompressing a document requires reading and decompressing all documents compressed before the document, leading to more I/O and computation. In WiSER, we trade space for less I/O by using more space but reducing the I/O while processing queries.

In addition, WiSER aligns compressed documents to the boundaries of file system blocks if the unaligned data would incur more I/O. A document may suffer from the block-crossing problem, where a document is unnecessarily placed across two (or more) file system blocks and requires reading two blocks during decompression. For example, a 3-KB data chunk has a 75% chance of spanning across two 4-KB file system blocks. To avoid this problem, WiSER aligns compressed document if doing so could reduce the block span.

3.5 Impact on Indexing

Our techniques focus on optimizing query processing instead of index creation since query processing is performed far more frequently. Overall, we believe the overhead introduced to indexing is more than justified by the significant performance improvements on query processing. Cross-stage data grouping does not add overhead to indexing since the same data is simply placed in different locations. Adaptive prefetching employs existing information and does not add any overhead during indexing. Trading space for less I/O adds moderate I/O overhead for the indexing phase (25% for Wikipedia) because the document store takes more space.

Building two-way cost-aware Bloom filters requires additional computation: the indexer in WiSER builds two Bloom filters, *set.before* and *set.after*, for each term in each document. Although a fixed number of hashing calls are required to add an entry to a filter and some filters are empty, the accumulative cost can be high. Currently, we have not optimized the building of Bloom filters. One way to speed up the building is to parallelize it, which also speeds up writing the filters to SSDs. Another way is to cache the hash values of popular terms to avoid hashing the same term frequently; popular terms appear hundreds of thousands times but would only need to be hashed once.

3.6 Implementation

We have implemented WiSER with 11,000 lines of C++ code, which allows us to interact with the OS more directly than higher-level languages. Data files are mapped by `mmap()` to avoid complex buffer management. We rigorously conducted hundreds of unit tests to ensure the correctness of our code.

The major implementation differences between WiSER and Elasticsearch are the programming languages and network libraries. From our experimentation, we found that C++ does not bring significant advantage to WiSER over Elasticsearch. In fact, to make the starting performance of WiSER similar to that of Elasticsearch we had to implement a number of optimizations: we switched from class virtualization to templates; we manually inlined frequently-called functions; we used case-specific functions to allow special optimizations for the case; and, we avoided frequent memory allocations (e.g., by reusing preallocated std::vector).

4 Evaluation

In this section, we evaluate WiSER with WSBench, a benchmark suite we built, which includes synthetic and realistic search workloads. The impact of a particular technique can be demonstrated by comparisons between two versions of WiSER (i.e., with and without the technique). For example, we demonstrate the effect of two-way cost-aware Bloom filters by comparing WiSER with and without them.

At the beginning of this section, we analyze in detail how each of the proposed techniques in WiSER is able to improve performance by significantly reducing read amplification. We show that: cross-stage data grouping reduces I/O traffic by 2.9 times; two-way cost-aware Bloom filters reduce I/O traffic by 3.2 times; adaptive prefetching prefetches only necessary data; and, trading disk space for less I/O reduces I/O traffic by 1.7 times.

Later in this section, we show that our techniques improve end-to-end performance. For example, we compare WiSER (with Bloom filters) and WiSER (without Bloom filters) to show that our Bloom filters increase query throughput up to 2.6x. We also show that WiSER delivers higher end-to-end performance than Elasticsearch, which indicates that WiSER is well implemented and its techniques can be applied to modern search engines.

We strive to conduct a fair comparison between Elasticsearch and WiSER. We pre-process the dataset using Elasticsearch and input the same pre-processed data to both WiSER and Elasticsearch. The pre-processor produces tokens, positions, and offsets. We implement the exact same relevance calculation (BM25 [7]) in WiSER as is used in Elasticsearch. The pre-processing and the implementation ensure that both WiSER and Elasticsearch will produce query results with the same quality. Despite our efforts, WiSER and Elasticsearch still have many differences (e.g., network implementation, where Elasticsearch performs poorly, and program languages). However, by comparing time-independent metrics such as read traffic size, we can see how WiSER reduces amplification, which in turn improves end-to-end performance.

We conduct experiments on machines with 16 CPU cores, 64-GB RAM and a 256-GB NVMe SSD (peak read bandwidth is 2.0 GB/s; peak IOPS is 200,000) [5]. We use Ubuntu with Linux 4.4.0. We optimize the configuration of Elasticsearch by following the best practices and tune parameters such as the number of threads, heap size and stack size.

To evaluate how well each search engine can scale up to large data sets that do not fit in main memory, our experiments focus on environments with a small ratio of main memory to working set size. The total size of English Wikipedia dataset is 18 GB, and from our experiments, we infer that the working sets are generally a few GBs. Therefore, we configure the search engine processes to use only 512 MB of memory (using a Linux container); this limits the engine's page cache to a small size (i.e., tens of MBs). Such a configuration allows us to demonstrate that our proposed techniques are effective at reducing read amplification, hiding I/O latency and increasing I/O efficiency, which are essential challenges at larger scale.

4.1 WSBench

We had to create our own benchmark to evaluate WiSER and Elasticsearch because existing benchmarks are not sufficient. To evaluate its engine, the Elasticsearch team uses Wikipedia [33,44,54] and scientific papers from PubMed Central (PMC); unfortunately, the Wikipedia benchmarks do not include real queries and the PMC dataset is very small (only 574,199 documents and 5.5 GB when compressed) with only a few hand-picked queries [8, 9].

We create WSBench, a benchmark based on the Wikipedia corpus, to evaluate WiSER and Elasticsearch. The corpus is from English Wikipedia in May 2018, which includes 6 million documents and 6 million unique terms (excluding stop words). WSBench contains 24 synthetic workloads varying the number of terms, the type of queries, and the popularity level of the queried terms (also known as document frequency: the number of documents in which a term appears). A high popularity level indicates a long postings list and large data size per query. These variables allow us to cover a wide range of query types and stress the system. WSBench also includes a realistic query workload extracted from production Wikipedia servers [21], and three workloads with different query types derived from the original realistic workload.

4.2 Impact of Proposed Techniques

We evaluate the proposed techniques in WiSER for three types of synthetic workloads: single-term queries, two-term queries, and phrase queries. Such evaluations allow us to investigate how the proposed techniques impact various aspects of the system as different techniques have different impacts on workloads. We investigate low-level metrics such as traffic size to precisely show why the proposed techniques improve end-to-end performance.

4.2.1 Cross-stage Data Grouping

Cross-stage grouping can reduce the read amplification for all types of queries. Here we show its impact on single-term and two-term queries where grouping plays the most important role; phrase queries are dominated by positions data where two-way cost-aware Bloom filters play a more important role (as we will soon show).

Figure 7 shows the decomposed read traffic for single-term queries. The figure shows that WiSER can significantly reduce read amplification (indicated by lower `waste` than Elasticsearch); the reduction is up to 2.9x. The reduction is more when the popularity level is lower because the block utilization is lower. To process queries with low-popularity terms, a search engine only needs a small amount of data; for example, an engine only needs approximately 30 bytes of data to

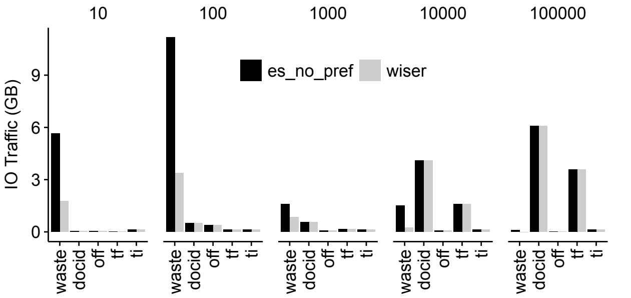


Figure 7: **Decomposed Traffic of Single-Term Queries** `waste` *represents the data that is unnecessarily read;* `docid`*,* `off`*,* `skiplist`*,* `tf`*, and* `ti` *represents the ideally needed data of document ID, offset, skip list, term frequency, term index/dictionary. Positions is not needed in match queries and thus not shown. This figure only shows the traffic from inverted index, which relates to cross-stage data grouping; we investigate the rest of the traffic (document store) later.*

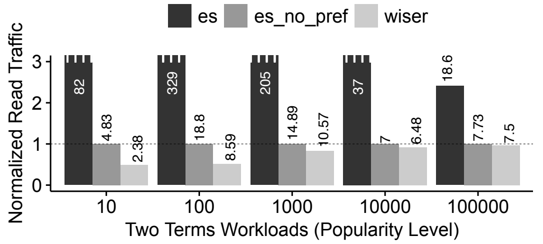


Figure 8: **I/O Traffic of Two-term Match Queries** *The size (GB) is normalized to the traffic size of Elasticsearch without prefetching.*

process the term "tugman" (popularity=8). To retrieve such small data, read amplification is inevitable as the minimal I/O request size is 4 KB. However, we can (and should) minimize the read amplification. Elasticsearch, which groups data by stages, often needs three separate I/O requests for such queries: one to term index, one to document IDs/term frequency, and one to offsets. In contrast, WiSER only needs one I/O request because the data is grouped to one block.

For high popularity levels (popularity=100,000), the traffic reduction is inconspicuous because queries with popular terms require a large amount of data for each stage (KBs or even MBs). In that case, the waste from grouping data by stages in Elasticsearch is negligible.

Figure 8 shows the aggregated I/O traffic for two-term queries, which read two postings lists. Similar to Figure 7, we can see that WiSER (`wiser`) incurs significantly less traffic than Elasticsearch. In this figure, we show two configurations of Elasticsearch: one with prefetch (`es`) and one without prefetch (`es_no_pref`). Prefetch is a common technique to boost performance in systems with ample memory; however, as shown in Figure 8, naive prefetch in Elasticsearch (`es`) can increase read amplification significantly. Such a dilemma motivates our adaptive prefetch.

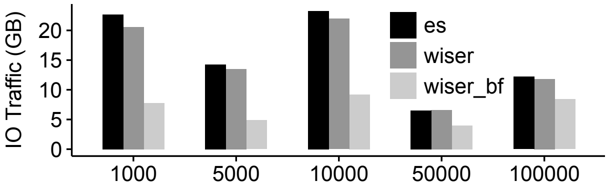


Figure 9: **I/O Traffic of Phrase Queries** *Results of Elasticsearch with prefetch is not shown as it is always much worse than Elasticsearch without prefetch.*

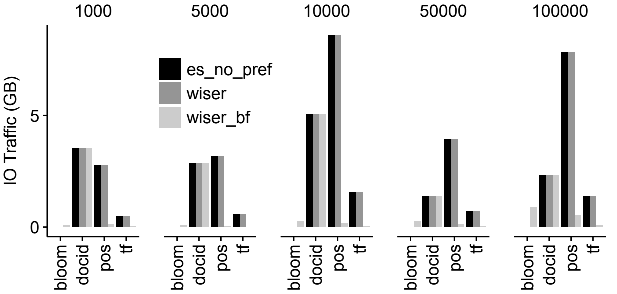


Figure 10: **Decomposed Traffic Analysis of Phrase Queries** *The bars show the ideal traffic sizes for each engine, assuming the storage device is byte-addressable. The sizes were obtained by adding counters to engine code.*

4.2.2 Two-Way Cost-Aware Bloom Filters

Two-way cost-aware Bloom filters only affect phrase queries as filters are only used to avoid positions data, which is used for phrase queries. In Figure 9, we show that WiSER without our Bloom filters demands a similar amount of data as Elasticsearch; WiSER with our Bloom filters incurs much less I/O traffic than WiSER without them and Elasticsearch.

Figure 10 shows the read amplification by the decomposed traffic in Elasticsearch, WiSER without Bloom filters, and WiSER with Bloom filters. The bars labeled with data type names show the data needed ideally, assuming the storage device is byte-addressable. First, we can see that applying filters significantly reduce the data needed ideally which is shown by the reduced aggregated size of all the bars. As shown in the figure, both Elasticsearch (`es`) and WiSER without filters (`wiser`) demand a large amount of position data; in contrast, WiSER with our two-way cost-aware filters (`wiser_bf`) significantly reduces positions needed in all workloads. Surprisingly, we find that our filters also significantly reduce the traffic from term frequencies (`tf`), which is used to iterate positions (an engine needs to know the number of positions in each document in order to iterate to the positions of the destination document). The traffic to term frequencies is reduced because the engine no longer need to iterate many positions.

Note that the introduction of our Bloom filters only adds a small amount of traffic to the Bloom filters (Figure 10), thanks to our two-way cost-aware design and the bitmap-based data layout of Bloom filters. The two-way cost-aware design allows us to prune by the smaller Bloom filter between the two Bloom filters of the two terms in the query. The bitmap-based

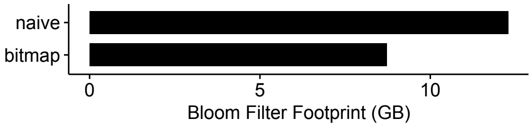


Figure 11: **Bloom Filter Footprint** *Our bitmap-based layout reduces footprint by 29%.*

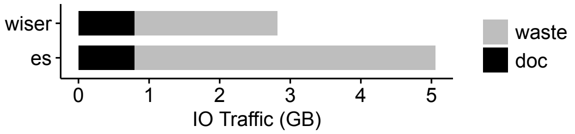


Figure 12: **Document Store Traffic** `doc` *indicates ideal traffic size. The relative quantity between Elasticsearch and WiSER is the same across different workloads; therefore, we show the result of one workload here for brevity (single-term queries with the popularity level = 10).*

layout, which uses only one bit to store an empty Bloom filter, significantly compresses Bloom filters, reducing traffic. We observe that 32% of the Bloom filters for Wikipedia are empty, which motivates the bitmap-based layout. Figure 11 shows that using bitmap-based layout reduces the Bloom filter footprint by 29%.

4.2.3 Adaptive Prefetching

Adaptive prefetching aims to avoid frequent wait for I/O and reduce read amplification by prefetching only the data needed for the current queries. As shown in Figure 7 and Figure 8, WiSER incurs less traffic than Elasticsearch with and without prefetching. As expected, by taking advantage of the information embedded in the in-memory data structure (Section 3.3), WiSER only prefetches the necessary data. Later in this section, we show that adaptive prefetching is able to avoid waiting for I/O and improve end-to-end performance.

4.2.4 Trade Disk Space for Less I/O

The process of highlighting, which is the last step of all common queries, reads documents from the document store and produces snippets. Figure 12 show that WiSER's highlighting incurs significantly less I/O traffic (42%) to the document store than Elasticsearch because in WiSER documents are decompressed individually and are aligned, whereas Elasticsearch may have to decompress irrelevant documents and read more I/O blocks due to misalignment. The size of WiSER's document store (9.5 GB) is 25% larger than that of Elasticsearch (7.6 GB); however, we argue that this space amplification is well justified by the 42% I/O traffic reduction. WiSER still wastes some traffic because the compressed documents in Wikipedia are small (average: 1.44 KB) but WiSER must read at least 4 KB (the file system block size).

4.3 End-to-end Performance

We examine various types of workloads in this section, including match queries (single-term and multi-term), phrase queries, and real workloads. For match queries, WiSER achieves 2.5 times higher throughput than Elasticsearch. For phrase queries, WiSER achieves 2.7 times higher throughput than Elasticsearch. WiSER achieves consistently higher performance than Elasticsearch for real-world queries. The end-to-end evaluation shows that WiSER is overall more efficient than Elasticsearch, thanks to our proposed techniques and efficient implementation.

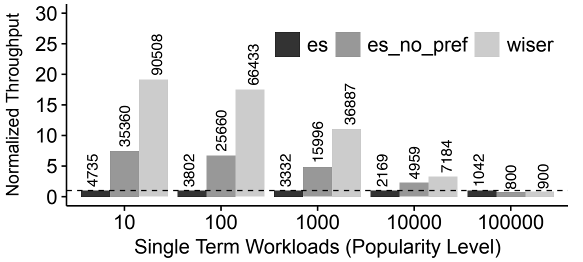


Figure 13: **Single Term Matching Throughput** *The throughput (QPS) is normalized to the performance of Elasticsearch with prefetch (the default 128 KB).*

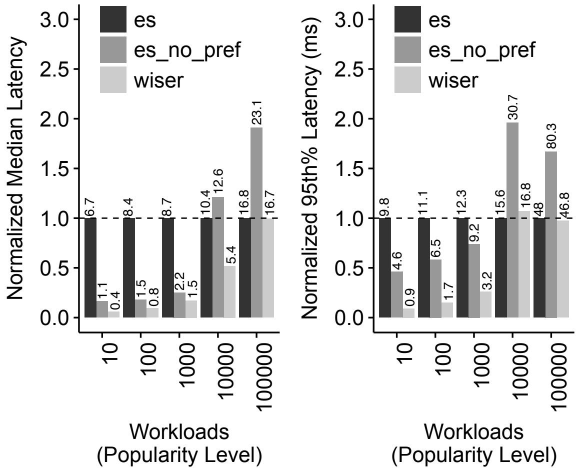


Figure 14: **Single Term Matching Latency** *The latency (ms) is normalized to the median latency of Elasticsearch with default prefetching.*

4.3.1 Match Queries

We now describe the results for the single-term and multi-term queries. Because queries that match more than two terms share similar characteristics with two-terms queries, we only present the results of two-term queries here.

Figure 13 presents the single-term match QPS (Queries Per Second) of WiSER. The default Elasticsearch is much worse than other engines when the popular levels are low because Elasticsearch incurs significant read amplification: Elasticsearch reads 128 KB of data when only a much smaller amount is needed (e.g., dozens of bytes). Elasticsearch without prefetch (`es_no_pref`) performs much better than `es_default`, thanks to much less read amplification.

WiSER achieves higher throughput than Elasticsearch without prefetch (`es_no_pref`) for low/medium popularity levels, which accounts for a large portion of the postings lists; the speedup is up to 2.5 times. When popularity level is 100,000,

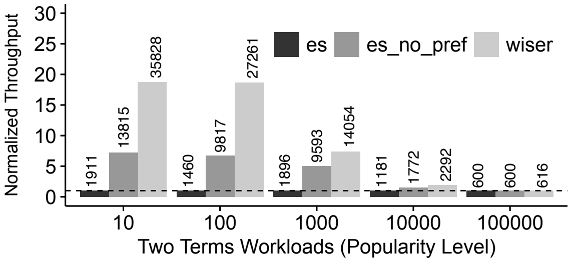


Figure 15: **Two Terms Intersecting Performance** *Throughput (QPS) is normalized to the performance of Elasticsearch.*

the query throughput of WiSER is 14% worse than Elasticsearch with default prefetching. We found that the difference is related to WiSER's less efficient score calculation, which is not directly linked to I/O.

The query throughput improvement largely comes from the reduced I/O traffic as queries with low popularity levels are I/O intensive and I/O is the system bottleneck. Indeed, we see that the query throughput improvement is highly correlated with the I/O reduction. For example, WiSER's query throughput for popularity level 10 is 2.6 times higher than Elasticsearch's; WiSER's I/O traffic for the same workload is 2.9 times lower than Elasticsearch's.

WiSER achieves better median latency and tail latency than Elasticsearch, thanks to adaptive prefetch and cross-stage data grouping. Figure 14 shows that WiSER achieves up to 16x and 11x lower median latency than Elasticsearch, for median and tail latency respectively. The latency of Elasticsearch is longer due to similar reasons for its low query throughput. Elasticsearch's data layout incurs more I/O requests than WiSER; the time of waiting for page faults adds to the query latency. In contrast, WiSER's more compact data layout and adaptive prefetch incur minimal I/O requests, eliminating unnecessary I/O wait.

Grouped data layout also benefits two-term match queries. Figure 15 presents results for two-term match queries, which are similar to single-term ones. As we have shown in Figure 8 that WiSER reduces by 17% to 51% of I/O traffic for workloads with popularity levels no more than 1,000. As a result, WiSER achieves 1.5x to 2.6x higher query throughput compared with Elasticsearch. When a workload includes popular terms, WiSER's traffic reduction becomes negligible since data grouping has little effects.

4.3.2 Phrase Queries

In this section, we show that our two-way cost-aware Bloom filters make fast phrase query processing possible. Specifically, WiSER can achieve up to 2.7 times higher query throughput and up to 8 times lower latency, relative to Elasticsearch. To support early pruning, WiSER needs to store 9 GB of Bloom filters (the overall index size increases from 18 GB to 27 GB, a 50% increase). We believe such space amplification is reasonable because flash is an order of magnitude cheaper than RAM.

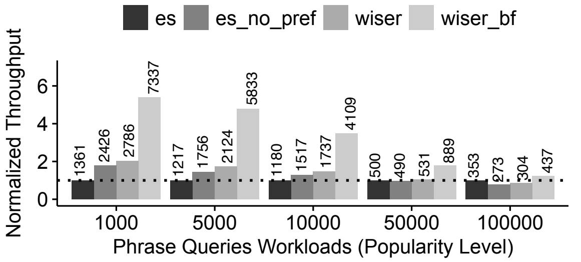


Figure 16: **Phrase Query QPS** *The throughput (QPS) is normalized to the performance of Elasticsearch.*

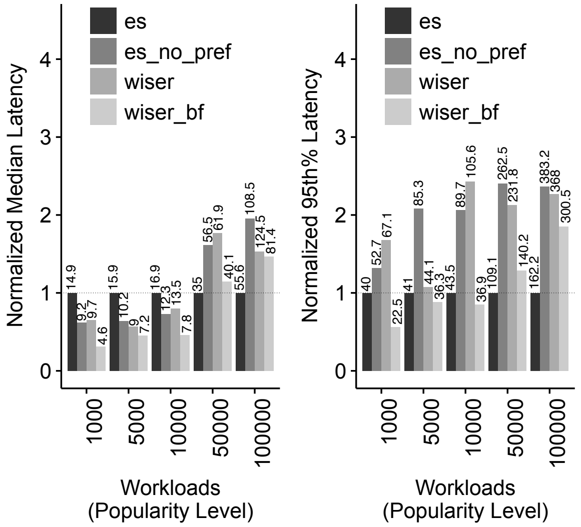


Figure 17: **Phrase Queries Latency** *The latency (ms) is normalized to the corresponding latency of Elasticsearch with default prefetching.*

WSBench produces the phrase query workloads by varying the probability that the two terms in a phrase query become a phrase. WSBench chooses one term from popular terms (popularity level is larger than 10,000); it then varies the popularity level of another term from low to high. As the popularity increases, the two terms are more likely to co-exist in the same document and also more likely to appear as a phrase in the document.

Figure 16 presents the phrase queries results among workloads in Elasticsearch (`es` and `es_no_pref`), WiSER (`wiser`), and WiSER with two-way cost-aware pruning (`wiser_bf`). The QPS of the basic WiSER (`wiser` and Elasticsearch with no prefetch (`es_no_pref`) are similar because our techniques in the basic WiSER (cross-stage data grouping, adaptive prefetch, trading space for less I/O) have little effect for phrase query. WiSER with two-way cost-aware Bloom filters achieves from 1.3x to 2.7x higher query throughput than that of basic WiSER, thanks to significantly lower read amplification brought by the filters.

Figure 17 shows that Bloom filters can significantly reduce latency (`wiser` vs. `wiser_bf`); also WiSER reduces median and tail latency by up to 3.2x and 8.7x respectively, compared to Elasticsearch. The reduction is more evident when

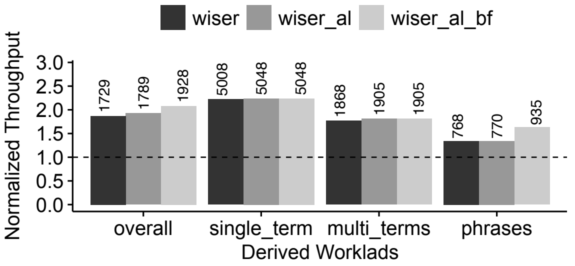


Figure 18: **Throughput of Derived Workloads** *The throughput (QPS) is normalized to the throughput of Elasticsearch without Prefetching*

the probability of forming a phrase is lower (low popularity level) because the Bloom filters are smaller in that case. Interestingly, Elasticsearch with OS prefetch (`es`) achieves the lowest latency when the probability of forming phrases is higher. The latency is lower because the OS prefetches 128 KB of positions data and avoids waiting for many page faults, although the large prefetch increases read amplification. In contrast, Elasticsearch without prefetch (`es_no_pref`) and WiSER do not prefetch; thus they may have to frequently stop query processing to wait for data (WiSER's adaptive prefetch does not prefetch position data due to fragments of unnecessary data.). However, the reduction of latency comes at a cost: although the latency of individual queries is lower, the query throughput is also lower due to the read amplification caused by prefetch (Figure 16).

Interestingly, we find our Bloom filters also speed up the computation of phrase queries. WiSER checks if a phrase exists by Bloom filters, which is essentially O(1) hashing. In the common case that the Bloom filter is empty, WiSER can even check faster because an empty Bloom filter is marked as 0 in the bitmap and we only need to check this bit. As a result, in addition to avoiding reading positions, Bloom filters also allow us to avoid intersecting the positions.

4.3.3 Real Workload

WSBench also includes realistic workloads. We compare Elasticsearch and WiSER's query throughput on the real query log; we also split the query log into different types of query workloads to examine the performance closely.

WiSER performs significantly better than Elasticsearch as shown in Figure 18. For example, for single-term queries, WiSER achieves as high as 2.2x throughput compared to Elasticsearch. We observe that around 60% queries in the real workload are of popularity less than 10,000, which benefits from our cross-stage data grouping. For multi-term match queries, grouped data layout also helps to increase throughput by more than 60%. For phrase queries extracted from the realistic workload, WiSER with Bloom filters increases throughput by more than 60% compared to Elasticsearch. Note that WiSER cannot achieve 2.7x higher throughout as shown in our synthetic phrase queries because, in this real workload, many phrases are the names of people, brand, or events and so on. Among these names, many terms are unpopular terms that are not I/O intensive, where pruning has limited effect. Finally, the overall performance of WiSER is similar to that of real single-term query log because single-term queries occupy half of the overall query log.

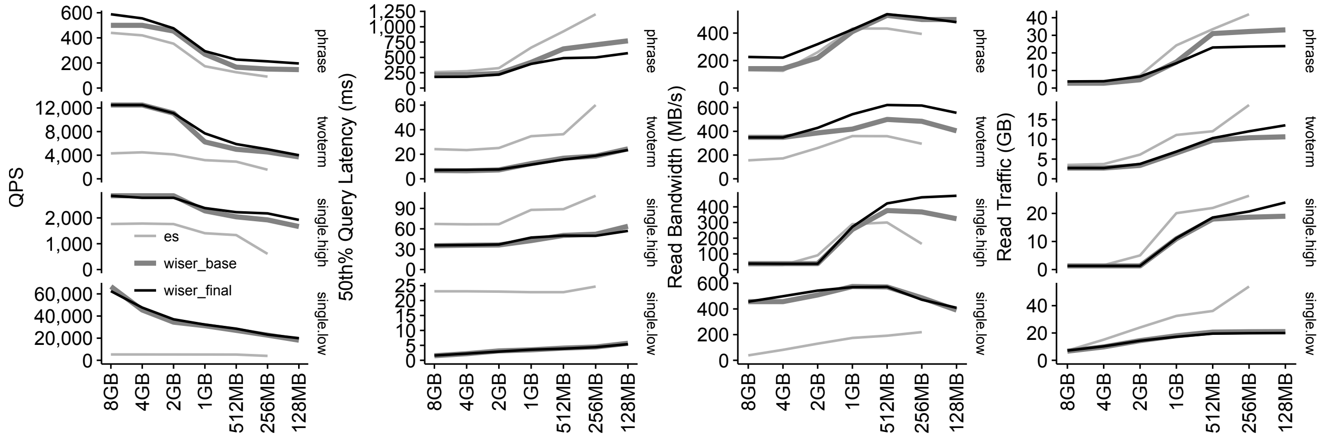
4.4 Scaling with Memory

Our previous experiments showed that WiSER performs significantly better than Elasticsearch for a single small memory size of 512 MB. This small memory size was chosen to stress the I/O performance of each search engine. For our final experiments, we show that we have rebuilt a search engine that can rely less on expensive memory and more on cheaper flash. Over a broader range of memory sizes, WiSER's techniques continue to improve I/O and end-to-end performance. In addition, it shows that WiSER works well with a low `memory size / working set size` ratio, which may allow WiSER to scale to large dataset without increasing much memory.

Figure 19 compares the query throughput, 50-th% query latency, bandwidth, and amount of traffic for Elasticsearch (`es`), WiSER with only cross-stage grouping (`wiser_base`), and fully-optimized WiSER (`wiser_final`). For our two end-to-end metrics, both versions of WiSER have much higher query throughput and much lower query latency than Elasticsearch across all workloads and memory sizes. As expected, query throughput is higher, and latency is lower, when more memory is available. For workload `twoterm`, `single.high`, and `single.low`, our highly efficient implementation allows WiSER with limited memory (e.g., 128 MB) to perform better than Elasticsearch with memory that can hold the entire working set (e.g., 8 GB).

The significant difference in end-to-end performance between WiSER and Elasticsearch for workload `twoterm`, `single.high`, and `single.low` at large memory sizes (Figure 19a and Figure 19b) is attributed to the network issue of Elasticsearch (we have carefully setup tests in Elasticsearch and compared it with similar applications to confirm the issue); with this memory size, I/O is not a bottleneck and our techniques should not make big differences. For the same workloads, we can identify the effects of our techniques in Figure 19d as we reduce the memory sizes, where I/O operations increase due to reduced cache. We can see that when the memory size is big, the traffic sizes between Elasticsearch and WiSER are similar, but WiSER's traffic sizes increase much slower than Elasticsearch's as we reduce memory sizes, thanks to data grouping, adaptive prefetching and trading disk space for I/O. Note that `wiser_final` has more read traffic than `wiser_base` because adaptive prefetch is turned on, which increases the read bandwidth (Figure 19c) and helps with end-to-end performance.

The effect of two-way cost-aware Bloom filters is evident in Figure 19a by comparing `wiser_base` (no Bloom filters) and `wiser_final`. The improvement is up to 1.4x (memory size = 256 MB, note that the improvement here is less than



(a) (b) (c) (d)

Figure 19: **Performance over a range of memory sizes.** *Elasticsearch fails to run at some low memory sizes and thus some of its data points are missing. Elasticsearch's default prefetch is turned off here because we have found that it hurts performance. Note that we place smaller memory sizes on the right side of the X axes to emphasize the effect of reducing memory sizes.*

the max in Figure 16 because we have queries with mixed popularity levels here). Figure 19d shows the `wiser_final`, the engine with two-way cost-aware Bloom filters, incurs much less I/O than `wiser_base` and `es` when memory is reduced; the reduction is also up to 1.4x (`wiser_base` vs `wiser_final`, memory size = 256 MB). As we can see, when I/O is the bottleneck, the reduction of traffic correlates well with improvement of end-to-end performance.

5 Related Work

Much work has gone into building flash-optimized key-value stores that utilize high-performance SSDs [23, 34, 38, 42]. For example, Wisckey [38] separates keys and values to reduce I/O amplification on SSDs. FAWN-KV [23] is a power-efficient key-value store with wimpy CPUs, small RAM and some flash. Facebook [34] proposes yet another SSD-based key-value store to reduce the consumption of DRAM by small block sizes, aligning blocks and adaptive polling.

Graph applications are also often optimized for SSDs. FlashGraph [59] speeds up graph processing by storing vertices in memory and edges in SSDs. MOSAIC [43] uses locality-optimizing, space-efficient graph representation on a single machine with NVMe SSDs. Many other work also facilitate high performance SSDs [48,49,60].

Search engines have different data manipulation and data structures from regular key-value stores and graphs. Among the limited literature, Wang et al. [55] and Tong et al. [52] studied studied search engine cache policies for SSDs; Rui et al. proposes to only cache metadata of snippets in memory and leave the data on SSDs because the I/O cost is reduced [58]. In this paper, we systematically redesign and implement many key data structures and processing algorithms to optimize search engines for SSDs. Such study exposes new opportunities and insights; for example, although using Bloom filters is straightforward in a key-value store, using them in search engines requires understanding the search engine pipeline, which leads us to the novel two-way cost-aware Bloom filter.

Many proposed techniques for search engines seek to reduce the overhead/cost of query processing [25–28,32,40,51]. These techniques may be adopted in WiSER to further improve its performance.

6 Conclusions

We have built a new search engine, WiSER, that efficiently utilizes high-performance SSDs with smaller amounts of system main memory. WiSER employs multiple techniques, including optimized data layout, a novel Bloom filter, adaptive prefetching, and space-time trade-offs. While some of the techniques could increase space usage, these techniques collectively reduce read amplification by up to 3x, increase query throughput by up to 2.7x, and reduce latency by 16x when compared to the state-of-the-art Elasticsearch. We believe that the design principle behind WiSER, "read as needed", can be applied to optimize a broad range of data-intensive applications on high performance storage devices.

Acknowledgments

We thank Suparna Bhattacharya (our shepherd), the anonymous reviewers and the members of ADSL for their valuable input. This material was supported by funding from NSF CNS-1838733, CNS-1763810 and Microsoft Gray Systems Laboratory. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] Apache Lucene. `https://lucene.apache.org/`.

[2] Apache Lucene Index File Formats. `https://lucene.apache.org/core/6_0_0/index.html/`.

[3] Apache Solr. `lucene.apache.org/solr/`.

[4] Breakthrough Nonvolatile Memory Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[5] CloudLab. http://www.cloudlab.us.

[6] DB-Engines Ranking. https://db-engines.com/en/ranking/.

[7] Elasticsearch. https://www.elastic.co/.

[8] Elasticsearch Adhoc Benchmark. https://elasticsearch-benchmarks.elastic.co/no-omit/pmc/index.html.

[9] Full Text Benchmark With Academic Papers from PMC. https://github.com/elastic/rally-tracks/blob/master/pmc/.

[10] Improving Readahead. https://lwn.net/Articles/372384/.

[11] LevelDB. https://github.com/google/leveldb.

[12] Libbloom. https://github.com/jvirkki/libbloom.

[13] Lucene Memory Index. https://lucene.apache.org/core/4_0_0/memory/org/apache/lucene/index/memory/MemoryIndex.html.

[14] Micron NAND Flash Datasheets. https://www.micron.com/products/nand-flash.

[15] RediSearch. redisearch.io/.

[16] RocksDB. https://rocksdb.org.

[17] Samsung 970 EVO SSD. https://www.amazon.com/Samsung-970-EVO-1TB-MZ-V7E1T0BW/dp/B07BN217QG/.

[18] Samsung K9XXG08UXA Flash Datasheet. http://www.samsung.com/semiconductor/.

[19] Samsung Semiconductor. http://www.samsung.com/semiconductor/.

[20] Toshiba Semiconductor. https://toshiba.semicon-storage.com/ap-en/top.html.

[21] WikiBench. http://www.wikibench.eu/.

[22] Wikimedia Moving to Elasticsearch. https://blog.wikimedia.org/2014/01/06/wikimedia-moving-to-elasticsearch/.

[23] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 1–14, Big Sky, Montana, October 2009.

[24] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[25] Nima Asadi and Jimmy Lin. Fast Candidate Generation for Two-phase Document Ranking: Postings List Intersection with Bloom Filters. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2419–2422, Maui, HI, 2012. ACM.

[26] Nima Asadi and Jimmy Lin. Effectiveness/efficiency Tradeoffs for Candidate Generation in Multi-stage Retrieval Architectures. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 997–1000, Dublin, Ireland, 2013. ACM.

[27] Nima Asadi and Jimmy Lin. Fast Candidate Generation for Real-time Tweet Search with Bloom Filter Chains. *ACM Transactions on Information Systems (TOIS)*, 31(3):13, 2013.

[28] Aruna Balasubramanian, Niranjan Balasubramanian, Samuel J Huston, Donald Metzler, and David J Wetherall. FindAll: a Local Search engine for Mobile Phones. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 277–288. ACM, 2012.

[29] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 359–374, Santa Clara, California, February 2017.

[30] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, pages 181–192, Seattle, Washington, June 2009.

[31] G. G. Chowdhury. *Introduction to Modern Information Retrieval*. Neal-Schuman, 2003.

[32] Austin T Clements, Dan RK Ports, and David R Karger. Arpeggio: Metadata Searching and Content Sharing with Chord. In *International Workshop on Peer-To-Peer Systems*, pages 58–68. Springer, 2005.

[33] Ludovic Denoyer and Patrick Gallinari. The Wikipedia XML Corpus. In *International Workshop of the Initiative for the Evaluation of XML Retrieval*, pages 12–19. Springer, 2006.

[34] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the EuroSys Conference (EuroSys '18)*, page 42, Porto, Portugal, April 2018. ACM.

[35] Evgeniy Gabrilovich and Shaul Markovitch. Computing Semantic Relatedness Using Wikipedia-based Explicit Semantic Analysis. In *IJcAI*, volume 7, pages 1606–1611, 2007.

[36] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the EuroSys Conference (EuroSys '17)*, pages 127–144, Belgrade Serbia, April 2017. ACM.

[37] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. *ACM Transactions on Storage*, 8(4):14, 2012.

[38] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 133–148, Santa Clara, California, February 2016.

[39] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 273–286, Santa Clara, California, February 2015.

[40] Jinyang Li, Boon Thau Loo, Joseph M Hellerstein, M Frans Kaashoek, David R Karger, and Robert Morris. On the Feasibility of Peer-to-peer Web Indexing and Search. In *International Workshop on Peer-to-Peer Systems*, pages 207–215. Springer, 2003.

[41] Wentian Li. Random Texts Exhibit Zipf's-law-like Word Frequency Distribution. *IEEE Transactions on information theory*, 38(6):1842–1845, 1992.

[42] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[43] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a Trillion-edge Graph on a Single Machine. In *Proceedings of the EuroSys Conference (EuroSys '17)*, pages 527–543, Belgrade Serbia, April 2017. ACM.

[44] David Milne and Ian H Witten. Learning to Link with Wikipedia. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 509–518. ACM, 2008.

[45] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[46] James K. Mullin. Optimal Semijoins for Distributed Database Systems. *IEEE Transactions on Software Engineering*, (5):558–560, 1990.

[47] Alexander Pak and Patrick Paroubek. Twitter as a Corpus for Sentiment Analysis and Opinion Mining. In *LREc*, volume 10, pages 1320–1326, 2010.

[48] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[49] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 472–488, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[50] Bin Tan and Fuchun Peng. Unsupervised Query Segmentation Using Generative Language Models and Wikipedia. In *Proceedings of the 17th international conference on World Wide Web*, pages 347–356. ACM, 2008.

[51] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. Efficient and Effective Retrieval Using Selective Pruning. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 63–72. ACM, 2013.

[52] Jiancong Tong, Gang Wang, and Xiaoguang Liu. Latency-aware Strategy for Static List Caching in Flash-based Web Search Engines. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1209–1212. ACM, 2013.

[53] Max Völkel, Markus Krötzsch, Denny Vrandecic, Heiko Haller, and Rudi Studer. Semantic Wikipedia. In *Proceedings of the 15th international conference on World Wide Web*, pages 585–594, 2006.

[54] Jakob Voß. Measuring Wikipedia. *Proceedings of ISSI 2005: 10th International Conference of the International Society for Scientometrics and Informetrics*, 1, 01 2005.

[55] Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. The Impact of Solid State Drive on Search Engine Cache Management. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 693–702. ACM, 2013.

[56] Fei Wu and Daniel S Weld. Autonomously Semantifying Wikipedia. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 41–50, 2007.

[57] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '19)*, Renton, WA, July 2019.

[58] Rui Zhang, Pengyu Sun, Jiancong Tong, Rebecca Jane Stones, Gang Wang, and Xiaoguang Liu. Compact Snippet Caching for Flash-based Search Engines. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1015–1018. ACM, 2015.

[59] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 45–58, Santa Clara, California, February 2015.

[60] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale Graph Processing on a Single machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, pages 375–386, Santa Clara, California, July 2015.