ISSN:0975-9646



# Sliced-scroll driven direct NVMe access for huge data pagination

Ruo Ando[#1]

[#]*National Institute of Informatics, Center for Cybersecurity Research and Development*
*Hitotsubashi, Chiyoda-ku, Tokyo 101-8430*

1ruo@nii.ac.jp

***Abstract*— As distributed data storage has become widely deployed, there is emerging demand of high performance data pagination for handling huge volumes over multiple shards.**

**Unfortunately, in contrast to recent rapid progress of rich-feature bulk/scroll API, there have been few research efforts on leveraging parallelism of NVMe SSD for building high performance application of data pagination to its full potential. We propose a new application of direct NVMe access driven by sliced scroll for huge data pagination of Elasticsearch. In our application, each task assigned with unique scroll ID directly accesses NVMe SSD with LBA calculation corresponding to the size of slice and JSON object. Besides, direct NVMe access is driven by sliced scroll with the 1-1 correspondence between slice and task. By doing this, our application can eliminate READ latency tremendously while evading the implementation and deployment cost. In experiment, we have adopted SPDK perf tool and measured READ/WRITE latency in huge data pagination of Elasticsearch. Specifically, READ latency reduction leads to a significant performance improvement in huge JSON objects parsing. It turned out that our SPDK based parser application can speed up the processing time compared with one of native Linux threads by ranging from x12.47 to x26.21 with the drastic reduction of READ latency**

***Keywords*— Sclied scroll, NVM Express, data pagination, Intel SPDK, user-mode I/O, Elasticsearch**

## I. INTRODUCTION

Scalability and capability to handle huge volumes of data in near real-time is emerging demand by many applications such as mobile apps, web, and data analytics applications.

In particular, concerning big data analytics applications of KV store and distributed data storage, data pagination has become key technology. For example, a user of KV store always needs more and more data either to render on a page or to process in the backend.Accordingly, in recent years, there have been rapid advances of helper API of deep pagination in both KV store and distributed data storage application such as Elasticsearch. However, on the other hand, existing distributed data storage applications cannot leverage NVMe SSD to its full potential for huge data pagination. From our operational experience of deploying distributed data storage [1], we have found two bottlenecks in huge data pagination.

### *A. Bottleneck1: READ latency in huge pagination*

1) Level-1 Heading*:* Data pagination is a functionality that is needed most of the time, for example, returning a large set of data to process in the backend or to simply re-index from one index to another. JSON based data pagination utility is alrealy adopted in popular KV store and distributed search engine such as Redis[11], MongoDB[12] and Elasticsearch[5] .

Recently, the size of JSON objects in distributed data storage is drastically increasing with an index of several millions, billions or sometimes trillions documents. Unfortunately, when trying to retrieve a large number of documents, you often see that when getting more and more with pages of the results, the queries slow down and finally timeout or result in memory issues.As a solution for this, Elasticsearch provides scroll-scan API[13] for data pagination without scoring[2].

However, even utilizing scroll API in WRITE phase, READ latency still remains serious pitfall in huge data paginaiton. Table 1 shows our brief measurement of the elapsed time of parsing paginated JSON objects with varying the size of JSON objects.In this measurement, we use conventional Linux I/O framework of fread() and Jansson [9]. For this measurement, we use a Linux machine equipped with Intel(R) Xeon(R) 6138 CPU @ 2.00GHz and 512 GB of memory running Ubuntu 16.04.

TABLE I
ELAPSED TIME (SEC) IN PARSING JSON OBJECTS WITH LINUX STANDARD I/O

| size of JSON objects | READ latency | Parsing |
|---|---|---|
| 3,039 MB | 52.03 | 0.184464 |
| 15,195 MB | 257.06 | 0.186884 |

In Table 1, the size of JSON objects is varied from about 3039 MB (1556 bytes * 2048 slices * 1000 lines) to 15195 MB (1556 bytes * 2048 slices * 5000 lines). In a nutshell, slice represents the number of JSON objects in one line of stream returned from Elasticsearch. As the size of JSON objects are increasing, it become more obvious that most of elapsed time is occupied by READ latency. The main reason is that the processing time of JSON parsing by Jansson keeps almost constant with around 0.18 sec with the size of JSON objects ranging from about 3,039 MB to 1,5195 MB, while READ latency is increased by about 5 times. From this measurement, we can obtain the insight that reducing READ latency is top priority to be considered for building high performance applica*tion using huge data pagination.*

### *B. Bottleneck2: Deployment cost of user-mode FS*

2) One way to eliminate the I/O stack overhead is to enable user processes to directly access storage devices

directly [7][4]. Because the kernel I/O stack accounts for a large fraction in total I/O latency and the high resource utilization of context switching. Consequently, some application using user-space I/O framework have been proposed to bypass the Linux block layer in high performance application [17]. Although user-mode I/O framework is effective in reducing I/O stack overheads, sometimes it imposes great burden on applications.

Particularly, user-mode file system and block device interface increase complexity in both driver and hardware implementations. These are useful in specific situation, but it often increases duplicate code across error-prone driver implementations, and usually nullifies generic features such as I/O scheduling and traffic shaping for QoS that are provided by a generic OS storage layer. It is expensive to make up for these utilities by additional implementations.

### C. Design goal

The design goal of our method is to reduce READ latency occurred in handling huge data pagination with simple and direct NVMe I/O submission. We adopt direct NVMe access for avoiding the deployment cost of user-mode file system and block device driver interface. Our design concept is that more directly the application access NVMe SSD, more effective the user-mode I/O framework is for the application of huge data pagination.

## II. Sliced scroll driven NVMe I/O submission

### A. Basic Design

We leverage parallelism of NVMe SSD for accelerating deep pagination of Elasticsearch. Figure 1 depicts our method. As we will illustrate the detail later, sliced scroll is multiple invocations of scroll API. In the upper side of Figure 1, Elasticsearch deploys M shards and splits M shards into N slices. For the response of bulk helper API of sliced scroll, Elasticsearch generates scroll IDs. Then, each task of SPDK of which ID is ranging from 1 to N takes unique scroll ID in the middle part of Figure 1. Task 1-N holds the same scroll ID during its operation of data pagination.

In other words, Task 1-N keeps retrieving JSON object repeatedly with their scroll ID until there is no more results left to return. At the lower side of Figure 1, in NVMe layer, LBA is calculated for each task to submit request I/O. LBA is calculated corresponding to the size of (1) slice and (2) JSON object. LBA is uniquely assigned to each task pipeline and incremented. The mechanism shown in Figure 1 is based on the 1 to 1 corresponding between task 1-N and slice 1-N by scroll ID.

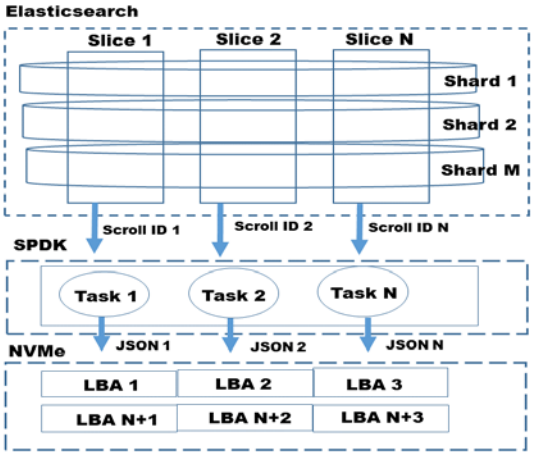


Fig. 1 Sliced scroll driven NVMe I/O submission.

### B. Scaling performance with sliced scroll

Elasticsearch provides a native API to scan and scroll over indexes using a cursor and you can scroll over it [13]. Scroll API is not designed for real-time requests, but rather for handling large volume of data, or in order to reindex the contents of one index into a new index with a different configuration or to process it in the backend.

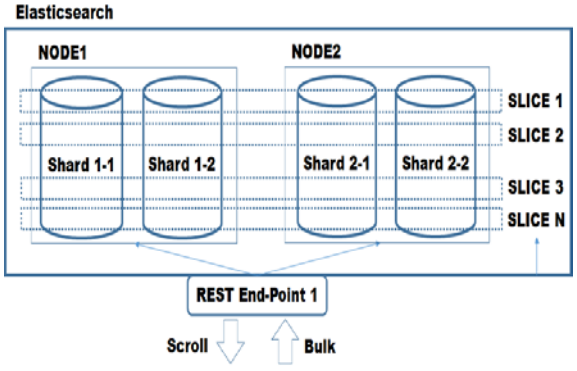


Fig. 2 Sliced scroll over multiple shards

Scroll queries can be further improved by the technique called as sliced scroll. By using sliced scroll, you can parallelize the scroll invocations. Figure 2 depicts how sliced scroll is built over multiple shards. Simply put, sliced scroll is a form of query with N split indices over 4 multiple shards (from shard 1-1 to 2-2). If you have N slices over 4 shards, as shown in Figure 2, you can parallelize single-threaded scroll (over 4 shards) with N threads. Instead of having a single thread to consume N * 4 hits, you can leverage the full computing power to consume those hits. From our experiment, the performance of sliced scroll can scale at least within 5-10 slices with CPU @ 2.00GHz and 512GB memory.

*B. Scaling performance with Intel SPDK*

Our tool implementation is based on SPDK perf [15]. Conventionally, the fio tool [6] is widely used because of its flexibility. However, that flexibility of fio causes overhead and reduces the efficiency of SPDK.

Accordingly, SPDK is released as a new benchmarking tool with minimal overhead during benchmarking. Figure 3 depicts the pipelines of task in our application based on SPDK.

SPDK perf tool adopts task parallelism for handling submission and completion queue. Generally, task parallelism means lots of different. In this case, independent tasks are working by splitting the data (the indices over multiple shards as shown in Figure 3) they consume. When you find task parallelism in SPDK perf, it is a special kind referred to as pipelining. Each pipeline is connecting I/O submission routine in the upper side of Figure 3 to completion callbacks in the lower side. One main difference between our application and SPDK perf is that each task 1-N is corresponding to slice ID 1-N. Also in the upper side of Figure 3, sliced_scroll() is invoked inside nvme_submit_io() for fetching JSON objects to be written to NVMe SSD.

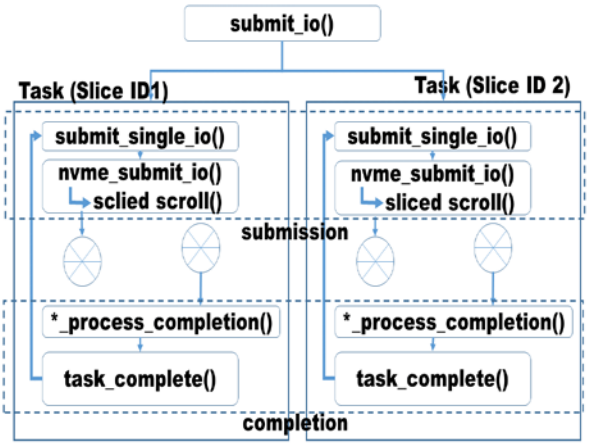


Fig. 3 Task parallelism of SPDK perf.

## III. Implementation in Detail

*A. Scroll ID and task allocation*

For scroll scan, Elasticsearch has a special endpoint for this API - the search/scroll endpoint. Endpoint returns the JSON object as follows:

```
{"_scroll_id":"XXXXXXXX","took":YYY,
"hits":{"hits":[{"_index":"ZZZ",_source":{[]}}}
```

Every call to the endpoint with scroll_id returns the next page of results. The _scroll_id section is a handle that we will take in the following queries. This object is used to get the actual data in subsequent requests. In the case that scroll queries cope with a large number of documents, it is possible to split the scroll in multiple slices which can be consumed independently. We implemented query builder/sender of scroll API by C Libcurl [8] as follows.

```
1: sprintf(post_data, "{"size": "%d",
"slice":"%d",
{"id":%d,"max":%d}}",
SLICE_SIZE, queue_depth,
slice_max);
2: struct curl_slist* headers = NULL;
3: headers = curl_slist_append(headers,
"Content-Type: application/json");
4: curl_easy_setopt(curl, CURLOPT_POSTFIELDS,
post_data);
```

At line 1, query string is built with three parameters: slice_id, slice_max and slice_size. The string of post_data at line 1 and 4 has three parameters as follows.

***ID (slice ID):*** The ID of the slice. The result from the first request returned documents that belong to the first slice (id=0)

and the result from the second request returned documents that belong to the second slice (id=1). Slice ID is unique to task ID.

***MAX (queue_depth):*** The maximum number of slices. Or the number of slices into which the task split the multiple shards.

***SIZE (SLICE_SIZE):*** The size parameter allows you to configure the maximum number of hits to be returned with each batch of results.

Once the scroll_id is obtained from endpoint of Elasticsearch, a task is newly allocated with unique scroll ID (_scroll_id) and it will repeat querying with its scroll ID until fetched results is empty. In other words, each call of the scroll API returns the next batch of results until there are no more results left to return. For embedding and activating scroll ID into the pipeline of SPDK perf tool, we have modified task structure as follows.

```
1: struct perf_task {
2: struct ns_worker_ctx *ns_ctx;
3: struct iovec iov;
4: struct iovec md_iov;
5: uint64_t submit_tsc;
6: bool is_read;
7: struct spdk_dif_ctx dif_ctx;
8: char scrollID[2048];
9: char JSON[SLICE_SIZE*JSON_SIZE];
10: int taskID;
11: #if HAVE_LIBAIO
12: struct iocb iocb;
13: #endif
14: };
```

We insert line 8-10 to handle queries / responses of the data pagination of Elasticsearch. At line 8, scrollID[2048] is an array for holding scroll ID uniquely assigned with each task during the pagination. Line 9 of JSON[SLICE_SIZE * JSON_SIZE] is a memory buffer

for storing JSON objects returned from Elasticsearch. Task is identified by taskID at line 10. By adding line 8 and 10, each task of SPDK perf can be driven by sliced scroll API with 1-1 correspondence between task and slice.

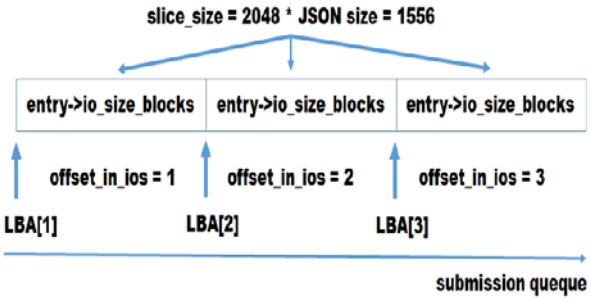


Fig. 4 Calculating LBA

*B. Calculating LBA*

In SPDK, the NVMe driver submits the I/O request with LBA as an NVMe submission queue using nvme_ns_cmd_xxx functions [15]. This function is executed in asynchronous manner and returns immediately before the command is completed.

```
1: rc = spdk_nvme_ns_cmd_write(ns_entry->ns,
2: ns_entry->qpair, sequence.buf,
3: N, /* LBA start */
4: 1, /* number of LBAs */
5: write_complete, &sequence, 0);
```

Here N at line 3 is the starting address of LBA, which is incremented as each task submits request I/O. Before calling nvme_ns_cmd_write(), LBA is calculated in nvme_submit_io() as follows. As shown at line 7, LBA is the product of offset_in_ios by entry->io_size_blocks.

```
1: static int
2: nvme_submit_io(struct perf_task *task,
3: struct ns_worker_ctx *ns_ctx,
4: struct ns_entry *entry,
5: uint64_t offset_in_ios)
5:{
6: uint64_t lba;
7: lba = offset_in_ios * entry->io_size_blocks;
```

Figure 4 shows the detailed illustration of the calculation of LBA. LBA is the smallest addressable data unit for READ and WRITE commands. LBA range is a collection of contiguous logical blocks specified by a starting LBA and number of logical blocks. More specifically, LBA is the product of offset_in_ios by entry->io_size_blocks as shown at line 7. The first term on right side, offset_in_ios, is incremented every time I/O submission is issued by each task. The second term, entry->io_size_blocks, represents LBA range which is the product of JSON_SIZE * SLICE_SIZE. For example, in the case of Figure 4, we obtain JSON object of which size is 1556 * 2048 = 3,186,688 bytes for every single I/O submission which means that entry->io_size_blocks is also 3,186,688 bytes.

*C. JSON objects parsing in READ task completion*

The application of SPDK adopts polled mode I/O completion on each queue pair to receive completion callbacks by calling spdk_nvme_qpair_process_completions().
SPDK perf tool has seven routines in I/O task completion starting from spdk_nvme_qpair_process_completions(). GDB stack trace of I/O task completion from spdk_nvme_qpair_process_completions() to task_complete() is as follows:

```
#0 task_complete (task)
#1 io_complete (ctx, cpl)
#2 nvme_complete_request (cb_fn<io_complete>,
cb_arg, qpair, req, cpl)
#3 nvme_pcie_qpair_complete_tracker (qpair,
tr, cpl, print_on_error=true)
#4 nvme_pcie_qpair_process_completions (qpair,
max_completions)
#5 nvme_transport_qpair_process_completions
(qpair, [max_completions)
#6 spdk_nvme_qpair_process_completions (qpair,
max_completions)
```

At frame #0, task_complete() takes the argument of struct perf_task discussed in section 2.2.1. Consequently, when the task_complete() is called at #0, we can obtain the buffer storing JSON objects by referring the argument of struct perf_task.

```
1: static inline void
2: task_complete(struct perf_task *task)
3: {
4: struct ns_worker_ctx *ns_ctx;
5: uint64_t tsc_diff;
6: struct ns_entry *entry;

7: if(!task->is_read) { /* WIRTE op */ }
8: else if(task->is_read) {
9: /* JSON stream parsing */
10: parse(task->dump)
11: submit_single_io(task);
```

The function of parse(task->dump) at line 10 is implemented for parsing JSON objects. After the parsing is finished, task_complete invokes submit_single_io(task) at line 11 for repeating the invocation of scroll API until there there are no more hits left to return.

## IV. Evaluation

We compare the performance results of our application based on SPDK with one of native Pthreads. For all the experiments, we use Dell PowerEdge R640, equipped with Intel(R) Xeon(R) 6138 CPU @ 2.00GHz and 512 GB RAM, running Ubuntu 16.04. All the performance evaluations are performed on a commercial Intel Optane SSD 905P 1.5TB NVMe SSD.

### *A. WRITE latency with scroll API*

We compare WRITE latency in two cases: POSIX Pthread using fwrite() and SPDK task using nvme_ns_cmd_xxx function. For simplicity (excluding the discussion about scalability about threads and tasks), we set the number of both of native threads and SPDK tasks to 2.

Then, the number of lines of all retrived data are set to 10,000. Finally, the size of one JSON object is set to 1556 bytes. Figure 5 shows the comparison of WRITE latency with varying slice size from 1,024 to 3,072. In WRITE phase, native thread (POSIX Pthread) is faster than SPDK tasks by about 50 - 60%.

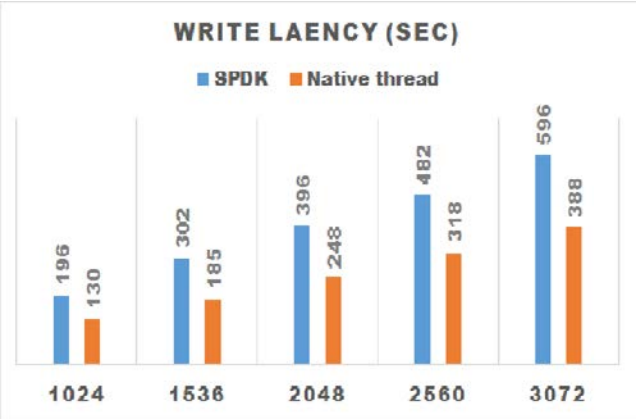


Fig. 5 WRITE latency (with varying slice size from 1024 to 3072 under 10,000 JSON objects).

### *B. READ latency with JSON parsing*

We measure the elapsed time of parsing JSON objects in using two interfaces: (1) direct READ access using SPDK and (2)fread() of Linux I/O interface. As with WRITE latency measurement, we set the number of both of native threads and SPDK tasks to 2. Also, we set the number of JSON_SIZE to 1556 bytes and slice sice to 2048. In READ phase, we vary the number of lines N ranging from 1,000 to 10,000. %, which means SLICE_SIZE is the parameter. Program fragment of JSON parsing is as follows. At line 2 and 3, I/O vector is transferred to JSON [SLICE_SIZE * JSON_SIZE] by strncpy(). At line 11-12, the program enters the loop for parsing JSON objects.

```
1: char JSON[SLICE_SIZE * JSON_SIZE];
2: strncpy(JSON, (char*)task->iov.iov_base,
3: SLICE_SIZE * JSON_SIZE);
4: json_error_t error;
5: json_t *result = json_loads(JSON, 0, &error);
6: json_t *repositories
7: = json_object_get(result, "hits");
8: json_t *value;
9: const char *key;
10: json_t *value_source;
11: json_object_foreach
12: (repositories,key,value){ ... }
```

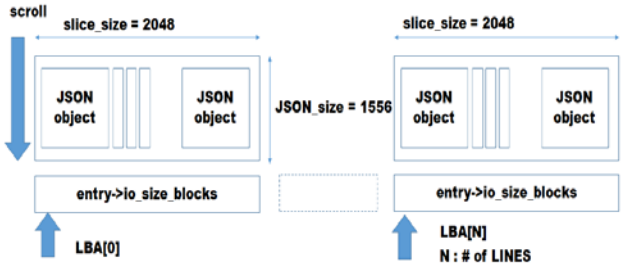


Fig. 6 Experimental setup in measuring READ latency (with 1,556 bytes under JSON object).

Figure 6 depicts our experimental setting of parsing JSON objects with NVMe direct access. For each I/O request of READ, we obtain 2048 (slice_size) JSON objects each of which size of 1,556 bytes. For example, if we get JSON objects with the number of lines 10,000, we fetched JSON objects of 1,556 (JSON size) * 2,048 (slice size) * 10,000 (lines) bytes (about 31,866 MB) after all I/O completion are done.

TABLE III
ELAPSED TIME (SEC) IN PARSING JSON STREAM(SEC)

| # of lines | Pthread(2) | SPDK task(2) | speed up |
|---|---|---|---|
| 1000 | 52.03 | 4.17 | x12.47 |
| 2000 | 102.87 | 7.32 | x14.05 |
| 3000 | 154.46 | 8.93 | x17.29 |
| 4000 | 205.65 | 10.11 | x20.32 |
| 5000 | 257.06 | 11.49 | x22.37 |
| 6000 | 303.96 | 12.28 | x24.75 |
| 7000 | 354.71 | 14.34 | x24.69 |
| 8000 | 400.45 | 16.92 | x24.39 |
| 9000 | 447.02 | 17.82 | x25.08 |
| 10000 | 503.17 | 19.17 | x26.21 |

Table 2 shows the elapsed time of parsing JSON objects.

We compare our SPDK based application using nvme_ns_cmd_xxx function with Pthread based application using fread(). For making the implication of the measurement clear, we run 2 native threads (Pthreads) and also 2 tasks (SPDK).

With the number of lines varying, we observe the drastic improvement of the processing time ranging from 12.47x (# of lines 1,000) to 26.2x (# of lines 10,000). This is because Direct NVMe access reduces the READ latency tremendously while the elapsed time of parsing keeps almost constant as shown in Table 1 in section 1.

## V. RLEATED WORK

There have been many research efforts on reducing the overheads the Kernel I/O stack. Shin et al. propose the multi-context I/O paths for SSD [14]. In [14], it is discussed that multi-context I/O paths can increase the I/O latency due to the overhead of context switching. Using polling instead of interrupts is another solution for removing context switching from the I/O path [3].

In the optimization of I/O stacks in kernel space, rling et al. [10] provide multiple queues on multi-cores to improve the I/O performance on NVMe SSD. Zhang

provides new I/O path optimization to minimize the overhead of I/O path for high priority tasks [19].

User-space filesystem is one of the most promising research fields. Moneta-D [4], based on Moneta [3] which is a flexible file-system architecture leveraging the storage-class memory, presents user-space software stacks to eliminate storage access latencies based on their own private and virtualized interface. Aerie[16] is a flexible user-level file system adopting the storage-class memory for user applications to access hardware without kernel interaction. EvFS[18] exposes asynchronous processing of complex file I/O with page cache and direct I/O for building a user-level storage stack.

## VI. Conclusion

Scalability and capability to handle huge volumes of data in near real-time is emerging demand by many applications such as mobile apps, web, and data analytics applications.

Concerning big data analytics applications of KV store and distributed data storage, data pagination has become key technology. In this paper, we have proposed sliced-scroll driven direct NVMe access for huge pagination. The main contributions of this paper as follows:

*1). We have addressed the bottleneck in huge pagination*

The most considerable bottleneck in huge data pagination is disproportional READ latency. READ latency reduction by our method leads to a significant performance improvement of
huge JSON objects parsing. It turned out that our SPDK based parser application can speed up the processing time compared with one of native Linux threads by ranging from x12.47 to x26.21 with the drastic reduction of READ latency.

*2). Direct access for minimizing the deployment cost*

The thrust of our method is that task assigned with unique scroll ID calculate LBA by itself and invokes nvme_ns_cmd_xxx function directly in user space. By doing this, NVMe I/O submission can be driven by sliced scroll API inside our user-space application with minimal implementation and deployment cost. It has become clear that more directly the application access NVMe SSD, more effective the user-mode I/O framework is for the application in huge data pagination.

## References


[1] Ruo Ando. Multi-gpu accelerated processing of timeseries data of huge academic backbone network in ELK stack. In 33rd Large Installation System Administration Conference (LISA'19), Portland, OR,October2019. USENIX Association.

[2] Marek Rogozinski Saurabh Chhajed. Bharvi Dixit, Rafal Kuc. Elasticsearch: A complete guide, chapter 7.

[3] De A. Coburn J. Mollow T. I. GUPTA R. K. Caufield, A. M. and S SWANSON. Moneta: A highperformance storage array architecture for next-generation, non-volatile memories. In In Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10), Atlanta, GA, USA, 2010.

[4] MollovT.I.EinserL.A.DeA.CoburnJ.Caufield,A.M. and S. Swanson. Providing safe, user space access to fast, solid state disks. In In Proceedings of ASPLOS (2012), 2012.

[5] Elasticsearch. https://github.com/elastic/elasticsearch.

[6] fioTool. https://fio.readthedocs.io/en/latest/fio_doc.html.

[7] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect:Auser-spacei/oframeworkforapplicationspecific optimization on nvme ssds. In 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16), Denver, CO, June 2016. USENIX Association.

[8] The libcurl API. https://curl.haxx.se/libcurl/c/.

[9] Jansson — C library for working with JSON data. http://www.digip.org/jansson/.

[10] DavidNellansMatiasBjørling,JensAxboeandPhilippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In In Proceedings of the 6th International Systems and Storage Conference (ACM), 2013.

[11] Redis: In memory data structure store. https://redis.io/.

[12] MongoDB. https://www.mongodb.com/.

[13] MarekRogozi´nski.RafałKu´c. Elasticsearchserver third edition, chapter 8.

[14] Chen Q. Oh M. Eom H. Shin, W. and H. Y. Yeom. Os i/o path optimizations for flash solid-state drives. In In Proceedings of USENIX Annual Technical Conference (USENIX ATC '14), Philadelphia, PA, USA, 2014.

[15] SPDK. https://spdk.io/.

[16] Nalli S. Pannerselvam S. Varadarajan V. Saxena P. Volos, H. and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In In Proceedings of Eurosys (2014), 2014.

[17] Minturn D. B. Yang, J. and F Hady. When poll is better than interrupt.In USENIX conference on File and Storage Technologies(FAST '12), San Jose, CA, USA, 2012.

[18] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. Evfs: User-level, event-driven file system for nonvolatile memory. In 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19), Renton, WA, July 2019. USENIX Association.

[19] JieZhang,Miryeong Kwon,Donghyun Gouk,Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI18),pages477–492,Carlsbad,CA,October2018. USENIX Association.