preadv (GNU/Linux) vs. readVectored (Hadoop Distributed File System) An Application Case Study

Arteen Abrishami

Abstract

The Hadoop Distributed File System is a distributed file system that operates using two key software abstractions: the `DataNode` and the `NameNode`. It is specifically designed for fault-tolerance, deployment on low-cost hardware, and high throughput access to large data sets. It is intrinsically designed for batch applications, not for those in need of low-latency access, and is the open-source sibling to the Google File System. Though non-POSIX compliant, this is solely intended to relax constraints to provide even higher throughput on streamed accesses. It targets consumers that require large data sets and is intended to support large files. In addition, it is intended to be portable, written in the Java programming language.

In this report, we compare an API available in Hadoop release 3.3.5, `readVectored`, to the Linux/GNU system call `preadv(2)`. `readVectored` was originally inspired by the Linux/GNU system call `preadv(2)`.

1 Overview of HDFS

Here we summarize HDFS [1].

Hadoop's `NameNode` and `DataNode` are pieces of software intended to run on any commodity hardware that supports the Java programming language. There is one `NameNode` in a cluster, intended to act as "master" and manage the file system namespace and regulate file accesses by clients. In addition, there is usually a number of `DataNodes` (can scale to hundreds) in a cluster that are intended to manage the storage attached to the nodes they run on. Files are split into one or more blocks (usually of large size) by the `NameNode`, being mapped to a set of `DataNodes`. In addition, they are replicated in a manner to support both fast transfer and fault tolerance, increasing availability of the data. The `NameNode` is responsible for metadata operations, while the `DataNodes` are responsible for directly serving read and write requests from clients so that there is no perceived bottleneck at the `NameNode`.

Each cluster maintains two `NameNode`s, one in an active state, and the other one on "standby". This, again, is to provide reliability and availability, so that the secondary `NameNode` can takeover at any time in the case of a failure. The namespace state is therefore fully synchronized between them.

In addition, the `NameNode` is responsible for managing the block replication policy, with this typically being done in to both increase throughput and reliability, with data being replicated in a "rack-aware" manner. In the default case, when the replication factor is 3, one replica is placed locally if the writer is a `DataNode` (or on the same rack if not), with the last two copies being placed on nodes on the same remote rack. Mainly, this is a write optimization, but it does not sacrifice read performance or data reliability.

The `readVectored` API, introduced in Hadoop release 3.3.5, is an extension to the `PositionedReadable` interface [3]. This is an operation for Vectored I/O, also known as Scatter/Gather I/O. We discuss this in detail below.

2 API Comparison

`readVectored(List<? extends FileRange> ranges, IntFunction<ByteBuffer> allocate)`: Java function call that takes in a set of ranges to read from and puts data into a set of buffers asynchronously, merges ranges based on values for `minSeekForVectorReads` and `maxReadSizeForVectorReads` [3, 4]

`preadv(int fd, const struct iovec *iov, int iovcnt, off_t offset)`: C standard library function that reads `iovcnt` buffers of data at the specified offset, where `iov` points to an array of `iovec` structures, containing the starting address of the buffer and the length in bytes to transfer [6, 7]

The two APIs are fundamentally different. `readVectored` is geared toward "seek-heavy" readers so that they can specify multiple ranges to read at once. This is not possible with `preadv(2)`. A similar paradigm (though not asynchronous) on a GNU/Linux system would be multiple calls to `pread(2)` (the non-vectored version of `preadv(2)`) with different buffers and offsets specified.

With regards to `preadv(2)`, the similarity they share ends at the fact that they are both reading and both specify multiple buffers to fill. `preadv(2)` functions identically to `pread(2)` (which also specifies a starting offset) but instead it fills multiple buffers in one contiguous streamed access to the file.

3 Feature Comparison

One advantage of both `readVectored` and `preadv(2)` is that the blocks of data being filled (in the application program) need not be contiguous in memory; they can be separately allocated.

This could also be seen as a disadvantage, depending on your perspective, making memory management that much more difficult. However, it also serves the advantage that you could potentially dispatch the work to be done for each filled buffer to a different thread to work on concurrently.

An advantage that `preadv(2)` provides is that the read is guaranteed to be atomic, in that it cannot be affected by reads from other threads or processes that have file descriptors pointing to the same open file description [7]. The documentation for `readVectored` [4] specifically states that some implementations may not guarantee positioned reads to be thread-safe, with streams in general not being thread-safe.

On the other hand, when `preadv(2)` executes, the task in question traps into the kernel, and since it is a blocking call, must wait until the request has been fulfilled. This could imply that the thread wait for many milliseconds in the case of worst delay. `readVectored` is asynchronous. This means that the thread (or task) may continue on with more useful work. However, one must consider that `readVectored` is layered upon HDFS, which is not necessarily low-latency, being a distributed file system.

Additionally, though a call to `pread(2)` is replaceable by a combination of other GNU/Linux system calls, it is difficult replace a call to `readVectored` in this manner. In order to do so, one must make call multiple calls to `pread(2)` with multiple, different buffers specified in order to simulate the ability that `readVectored` has to read in different "ranges". However, the calls would be synchronous and (in our example) single-threaded, whereas `readVectored` would return immediately, handle all that work for you and potentially perform the reads in *parallel* using the underlying HDFS system. It should be considered that `readVectored` will speak directly to the `DataNodes`, so that there is no bottleneck at the `NameNode`.

4 Failure Modes

The implementation for `preadv(2)` may fail for any of the reasons that a `read(2)` or `lseek(2)` may fail, in addition to an errno of `EINVAL` if the sum of `iov_len` overflows a `ssize_t` value, or if the vector count `iovcnt` is less than zero or greater than the permitted maximum (1024 on Linux). [6]

Some example reasons for failure are:

1. invalid file descriptor
2. buffer address outside accessible range
3. interrupted by signal before data read
4. I/O error
5. file descriptor refers to a directory or other non-allowed file type
6. invalid seek
7. offset not representable in `off_t`

Based on this information, we can see several of these failure modes are not possible with the Java implementation of `readVectored`. This is because the "file descriptor" is intrinsically linked to the abstract `FSDataInputStream`. A key assumption here is that the "stream being read references a finite array of bytes" [5]. That means that things like (1) and (5) are intrinsically not possible, since the file descriptor is already established. Additionally, Java memory management does not allow the usage of explicit pointers, so something like (2) is not possible either.

However, something like (4) is possible in both, but could possibly be more likely with something like `readVectored` since there are more components in HDFS than a typical GNU/Linux file system, meaning that there are naturally more things could go wrong during the I/O process. HDFS aims to provide high availability and reliability, so though unlikely, an I/O error is much more likely (multiple node failures, network down, congested routers, etc.).

Additionally, since the call to `readVectored` returns multiple buffers asynchronously, it is completely possible that some may be filled with valid data. Since each of the ranges comes back with a `CompletableFeature` that tells you when the data for that region is read, you could potentially get a partial read of some buffers, even upon a failure [5].

5 Example Application

`readVectored` would benefit big-data computation that requires high-throughput access to lots of data, at different offsets within the overall data structure. The nature of its asynchronous call allows the application to process some buffers in parallel while others are being fetched, and it provides simplicity where one call to `readVectored` may replace multiple `pread(2)` calls, as we mentioned before. On the other hand, `preadv(2)` simply allows contiguous data to be offloaded into multiple buffers in memory.

Therefore, `readVectored` may be suitable for an application that (1) operates on huge amounts of data and would benefit from cloud computing/a distributed network (2) performs seeks often and heavily (3) would sacrifice latency at the cost of throughput. An example application is a web crawler or indexer, like the one that Google uses, which may want to read in many ranges of data all in parallel so that it can go through and produce the necessary computations.

On the other hand, something like `preadv(2)` may be beneficial to an application already running on a GNU/Linux system that simply wishes to batch a contiguous read at a specific offset into multiple buffers.

For both, the batching of data into multiple buffers could be done to allow multiple threads to then process the data in parallel, for example. This is an example of bulk synchronous parallel computing. A potential use case in POSIX-compliant system could be to read data from a large file, dispatch to threads via a call to `pthread_create`, which then hash the values into a histogram (using per-bucket locks).

Overall, by its nature, Vectored I/O is designed with batch-processing in mind, but due to the differences in implementation, and the underlying system support, the use cases for `preadv(2)` and `readVectored` vary significantly.

References

[1] Apache Software Foundation. *HDFS Architecture*. 2023. URL: https://hadoop.apache.org/docs/r3.3.5/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[2] Apache Software Foundation. *Hadoop Release 3.3.5*. 2023. URL: https://hadoop.apache.org/release/3.3.5.html.

[3] Apache Software Foundation. *Hadoop 3.3.5 Overview of Changes*. 2023. URL: https://hadoop.apache.org/release/3.3.5.html.

[4] Apache Software Foundation. *Hadoop 3.3.5 readVectored API*. 2023. URL: https://hadoop.apache.org/docs/r3.3.5/hadoop-project-dist/hadoop-common/filesystem/fsdatainputstream.html.

[5] Mukund Thakur. "Hadoop Vectored IO: your Data just got Faster!" ACNA. 2022. URL: https://www.apachecon.com/acna2022/slides/02_Thakur_Hadoop_Vectored_IO.pdf.

[6] Linux man-pages project. *preadv2(2) — manpages-dev — Debian testing — Debian Manpages*. man-pages release 6.02. URL: https://manpages.debian.org/testing/manpages-dev/preadv2.2.en.html.

[7] Linux man-pages project. *preadv(2) - Linux man page*. URL: https://linux.die.net/man/2/preadv.