Hadoop Vector IO: cloud IO for columnar data formats

STEVE LOUGHRAN, MUKUND THAKUR, and OWEN O'MALLEY

The Hadoop Filesystem API has been extended to support scatter/gather IO for columnar data formats. This paper describes the design and implementation of this feature, and the integration with Apache ORC and Parquet, and presents results from TPC-DS benchmarks.

Or key findings are that

- An asynchronous scatter/gather IO API is ideally suited to retrieval of "stripes"/"row groups" of data stored in columnar data formats.
- Java's native IO APIs support such an API, offering speedups when reading local data.
- Against cloud storage, the ability to issue parallel GET requests significantly improves read performance, providing tangible performance improvements to queries which read large quantities of data.
- Using the API in the ORC and Parquet is sufficient to enable speedups in applications using the libraries *without* any changes in the application code.
- The API shows that "cloud first" APIs provide tangible speedups against the classic Posix API. Many more opportunities exist to provide and exploit such APIs.

ACM Reference Format:
Steve Loughran, Mukund Thakur, and Owen O'Malley. 2024. Hadoop Vector IO: cloud IO for columnar data formats. 1, 1 (April 2024), **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.0.1 Terminology. First, some terminology needs to be introduced to describe the protocols.

2 THE CHALLENGE OF OBJECT STORES

Having introduced the classic filesystem and the commit protocols and algorithms used to commit the output of distributed computation, let us consider Object Stores such as Amazon S3, Google Cloud Storage and Windows Azure Storage.

The most salient point, is this: Object Stores are not filesystems. Rather than the classic hierarchical view of directories, subdirectories and paths, object stores store a set of objects, each with a unique key; a sequence of characters provided when the object was created. Classic path separators "/" are invariably part of the set of valid characters, so allowing objects to be created which have the appearance of files in a directory.

As examples, the following are all valid keys on the Amazon, Google and Microsoft stores

```
/entry
/path1/path2/path3
/path1/
/path1
```

More subtly, it is valid for an object store container (on S3:, a "bucket") to have objects with all of these names simultaneously. It is not an error to have an object whose key would make it appear to be "under" another object, nor to explicitly contain path entries separators.

Authors' address: Steve Loughran; Mukund Thakur; Owen O'Malley.

2024. Manuscript submitted to ACM

Objects cannot generally be appended to once created, or renamed. They can be replaced by new objects or deleted. Some form of copy operation permits an object to be duplicated, creating a new object with a different key. Such copy operations take place within the storage infrastructure with a copy time measurable in megabytes/second.

The set of operations offered are normally an extended set of HTTP verbs:

PUT	Atomic write of an object
GET	retrieve all or part of an object
HEAD	retrieve the object metadata
LIST	list all objects starting with a given prefix
COPY	copy a single object within the store, possibly from other containers.
DELETE	Delete an object

3 DESIGN

The design of the vector IO API had some core requirements:

- Support scatter/gather IO through local filesystem through the java NIO API.
- Local reads to support with CRC validation of data read.
- Efficient support against cloud stores where parallel GET requests are required.
- Support scatter/gather IO through HDFS and other distributed filesystems where the POSIX API and Hadoop PositionedReadable interface are available
- Optimize for reading columnar data formats such as ORC and Parquet
- Be easy to integrate with the ORC and Parquet libraries, in the open source releases, as well as our own internal branches.
- Permit out-of-order result processing, even while the current format libraries do not yet support this.

What is key is: it *must* be possible to use the vector IO API against any existing Hadoop input stream, with the base implementation falling back to the existing `PositionedReadable` readFully() method, with the option of custom higher performance implementations — including for the local filesystem as well as cloud storage.

Java NIO API:

extending PositionedReadable with range coalescing

Async API with back-references for wiring up.

S3A Impl as ranged GET requests with (limited) parallelism.

4 IMPLEMENTATION

4.1 Default implementation

The default implementation of the vector IO API uses blocking readFully() calls for each range. This has the same performance as the existing read() API.

4.2 Local/Checksum filesystem implementation

The local filesystem implementation uses the Java NIO API to read the data into on-heap buffers in parallel.

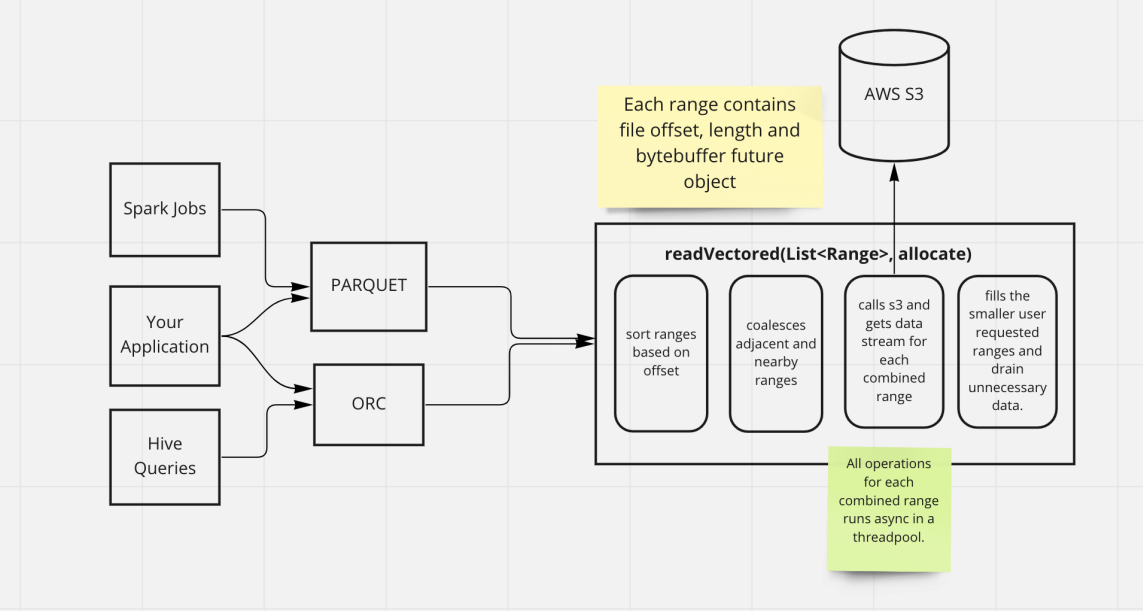

Fig. 1. S3A Vector IO

4.3 S3A

S3A implementation of vectored IO coalesces the nearby ranges into a multiple combined ranges and issues parallel GET requests to AWS S3. Diagram below shows the full workflow.

5 INTEGRATION WITH ORC AND PARQUET

It was our belief that integration with the ORC and Parquet libraries was sufficient to deliver tangible speedups

5.1 ORC

5.2 Parquet

6 RESULTS

6.1 External tests

Some numbers from an independent benchmark. I used Spark to parallelize the reading of all rowgroups (just the reading of the raw data) from TPC-DS/SF10000/store_sales using various APIs.

I used a modified (lots of stuff removed) version of the ParquetFileReader and a custom benchmark program that reads all the row groups in parallel and records the time spent in each read from S3. The modified version of ParquetFileReader can switch between the various methods of reading from S3. The entry AWS SDK V2 is a near copy of the Iceberg S3FileIO code though.

```
32 executors, 16 cores
fs.s3a.threads.max = 20
```

Reader	Mean Time/min)	Median	Baseline
Parquet	10.32	10	1
Parquet Vector IO	2.02	2	5.1
SDK V2	9.86	10	1
SDK V2 Async	9.66	9.6	1.1
SDK V2 AsyncCrt	9.76	10	1.1
SDK V2 S3TransferManager	9.58	9.5	1.1
SDK V2 Async CRT Http Client	10.8	11	1

Table 1. Independent benchmark results

I saw issues with the CRT client when running at scale causing JVM crashes. And the V2 transfer manager did not do range reads properly.

Summary - The various V2 SDK clients provide lower latency and better upload speeds but for raw data scans, they are all pretty much the same. Increasing the parallelism as vector IO does, has maximum benefit.

Analysis.

This benchmark was done with a Hadoop 3.3.x release which had not yet moved to the AWS SDK V2; this is why the baseline 10:32 time is slower than the "AWS SDK V2" timing. Now that Hadoop 3.4.x has moved to the AWS SDK V2, we would expect the v2 timings to be the default, so the speedups from other methods would not be quite so pronounced.

The AWS V2 SDK includes an "asynchronous client" which is only used by the S3A connector when copying or uploading files, not when reading them. The S3TransferManager is the class used to manage these operations. The "CRT http client" is a native C library to which the SDK can delegate much of the work of building, signing, sending and receiving HTTP requests, and processing the responses.

Although the CRT promises speedups, our experience with using openssl instead of the Java JRE's open TLS implementation makes us wary of the potential for unusual failure conditions which may occur in some deployments.

For the open source hadoop releases, moving to the "Async" client is something which could be done with lower risk; switching to CRT library would then be an optional which could be explicitly enabled in deployments.

7 LIMITATIONS

8 RELATED WORK

9 CONCLUSIONS AND FURTHER WORK

9.0.1 Multi-ranged GET requests. .

The HTTP protocol allows for a client to request multiple ranges of a file in the same request. If supported by cloud object stores, this will allow for a single request over a single HTTP connection to retrieve different row groups in the same request, *irrespecive of their location in the file.* This would assist coalescing range requests, reduce the number of HTTP requests needed while avoiding the need to discard data between ranges.

9.0.2 Footer prefetching and cacheing. As shown in

When these files are first opened, the initial read sequence is

- open the file
- read the final 8 bytes to validate filetype and the offset for the "real" footer

- read the full footer

On HDFS the costs of this are

- namenode: open the file
- read the final 8 bytes to validate filetype and the offset for the "real" footer
- read the full footer

ACKNOWLEDGMENTS

We are grateful for the contributions of all reviewers and testers of this work, especially our fellow developers in the open source community, especially Dongjoon Hyun, Parth Chandra, and Gang Wu. Special mention must be made of the Cloudera QE team whose scale testing not only produced the numbers in this paper, but also identified a large number of regressions related unrelated changes in the S3A codebase (move to AWS v2 SDK). As much of that work was contributed by Mukund and Steve, we are grateful.

10 REFERENCES

REFERENCES

A APPENDIX

We like to introspect and review what problems actually occurred in production. This helps us review our assumptions, consider how valid they proved to be, and to let us analyze how they got through our reviews and testing. Hopefully others can learn from our mistakes.

A.1 HADOOP-19101. Vectored Read into off-heap buffer broken in fallback implementation

Default implementation reads into Direct (off JVM-heap buffer reads broken. This surfaced when adding tests for azure storage. Our previous tests against the local and S3 stores used custom implementations which were correct. Lesson: always test those defaults. This has never been seen “in the wild” because the applications using the library do not seem to be reading into Direct buffers. As well as fixing the bug, the integration patch for Parquet reports all attempt to use vectored IO with Direct buffers as “unsupported”. This ensures that even when using old releases, the bug will not surface.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.