Hadoop Vectored IO:
your Data just got Faster!

Mukund Thakur
Software Engineer @Cloudera
mthakur@apache.org
Committer@Apache Hadoop

CLOUDERA

APACHECON North America

Hadoop Vectored IO: your data just got faster!

Not just faster, cheaper as well.

Everybody hates slow!

- Slow running jobs/queries.
- Slow websites.
- Slow computer.
- .....

Vectored IO api; your reading ORC and Parquet will get faster.

Agenda:

- Problem and solution.
- New vectored IO api spec.
- Implementation details.
- How to use?
- Benchmarks.
- Current state.
- Upcoming work.

seek() hurts

- HDDs: disk rotations generate latency costs
- SSDs: block-by-block reads still best
- Cloud storage like S3 or Azure: cost and latency of GET requests against new locations
- Prefetching: connector code assumes sequential IO
- ORC/Parquet reads are random IO, but can read columns independently, with locations inferred from file and stripe metadata

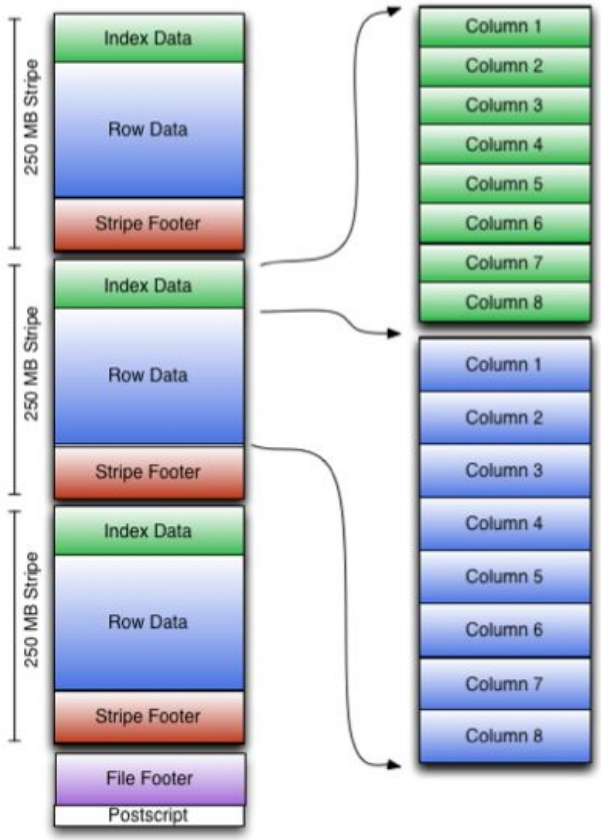
If the format libraries can pass their "read plan" to the client, it can optimize data retrieval

GET requests on remote object store: costly and slow.

Iostats for a task while running tpcds Q53.

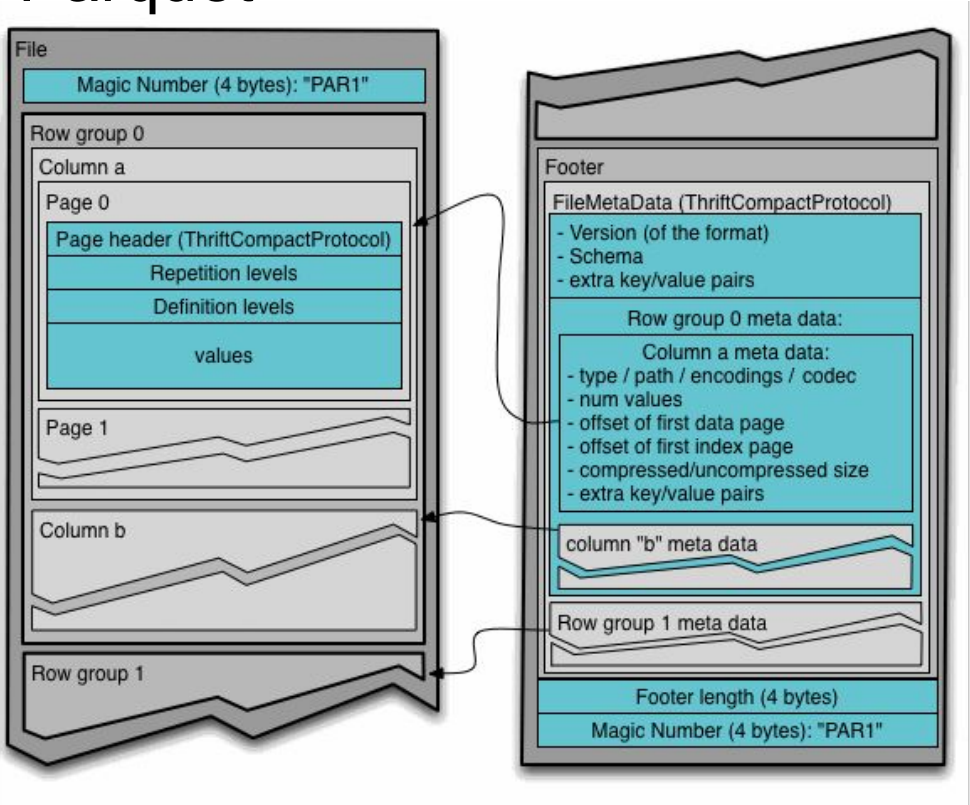
```
counters=(

(action_http_get_request=30) (stream_read_remote_stream_drain.failures=0)
(stream_read_close_operations=10) (stream_read_bytes=2339593)
(stream_read_bytes_backwards_on_seek=16068978) (stream_read_seek_bytes_discarded=16146)
(action_file_opened=10) (stream_read_opened=30) (stream_read_fully_operations=50)
(stream_read_seek_backward_operations=5) (stream_read_closed=30)
(stream_read_seek_operations=40) (stream_read_total_bytes=2685289)
(stream_read_bytes_discarded_in_close=329614) (stream_write_bytes=2751))

means=((action_file_opened.mean=(samples=10, sum=414, mean=41.4000))

(action_http_get_request.mean=(samples=30, sum=2187, mean=72.9000)));
```

ORC



- Postscript with offset of footer
- Footer includes schemas, stripe info
- Index data in stripes include column ranges, encryption, compression
- Columns can be read with this data
- Predicate pushdown can dynamically skip columns if index data shows it is not needed.

Parquet



- 256+ MB Row Groups
- Fixed 8 byte postscript includes pointer to real footer
- Engines read/process subset of columns within row groups

Solution: Hadoop Vectored IO API

- A vectored read API which allows seek-heavy readers to specify multiple ranges to read.
- Each of the ranges comes back with a CompletableFuture that completes when the data for that range is read.
- Default implementation reads each range data in buffers.
- Object store optimizations.
- Inspired from readv()/writev() from linux os.

Works great everywhere, radical benefit in object stores

New method in PositionedReadable interface:

```
public interface PositionedReadable {
 default void readVectored(
    List<? extends FileRange> ranges,
    IntFunction<ByteBuffer> allocate)
     throws IOException {
  VectoredReadUtils.readVectored(this, ranges, allocate);
 }
}
```

Hints for range merging:

- `minSeekForVectorReads()`

  The smallest reasonable seek. Two ranges won't be merged together if the difference between end of first and start of next range is more than this value.

- `maxReadSizeForVectorReads()`

  Maximum number of bytes which can be read in one go after merging the ranges. Two ranges won't be merged if the combined data to be read is more than this value. Essentially setting this to 0 will disable the merging of ranges.

Advantages of new Vectored Read API:

Asynchronous IO: Clients can perform other operations while waiting for data for specific ranges.

Parallel Processing of results: Clients can process ranges independently/Out of Order

Efficient: A single vectored read call will replace multiple reads.

Performant: Optimizations like range merging, parallel data fetching, bytebuffer pooling and java direct buffer IO.

Implementations:

Default implementation:

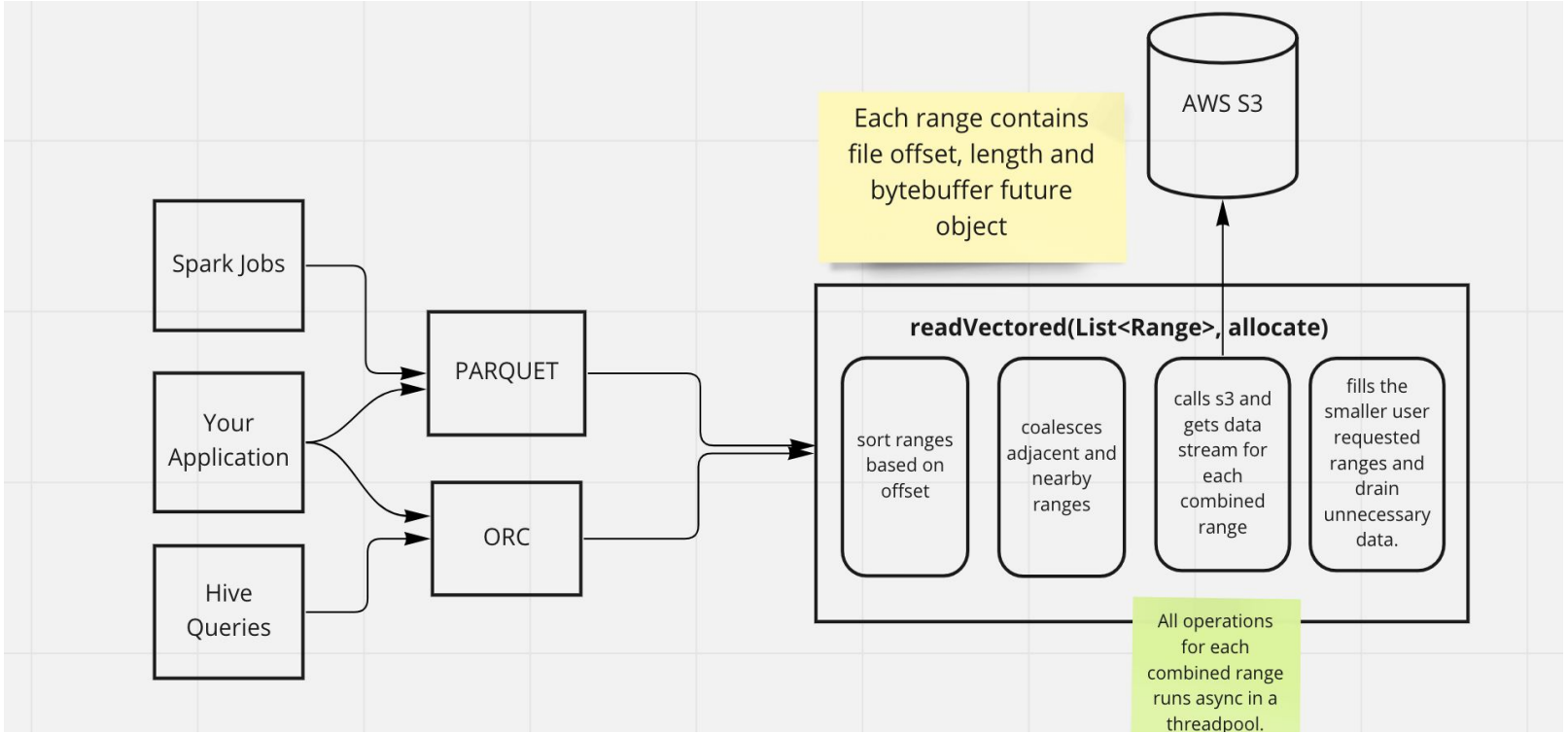
Blocking readFully() calls for each range data in buffers one by one.

It is exactly same as the using the old read APIs.

Local/Checksum FileSystem:

Merge the nearby ranges and uses java Async channel api to read combined data into large byte buffer and return sliced smaller buffer for each user ranges.

S3A parallelized reads of coalesced ranges



Features in S3A:

- Supports both direct and heap byte buffers.
- Introduced a new buffer pool with weak referencing for efficient garbage collection for direct buffers.
- Terminates already running vectored reads when input stream is closed or `unbuffer()` is called.
- Multiple get requests issued into common thread pool.
- IOStatistics to capture Vectored IO metrics.
- If/when s3 GET supports multiple ranges in future, could be done in a single request.

How to use?

```
public void testVectoredIOEndToEnd() throws Exception {
 FileSystem fs = getFileSystem();
 List<FileRange> fileRanges = new ArrayList<>();
 fileRanges.add(FileRange.createFileRange(8 * 1024, 100));
 fileRanges.add(FileRange.createFileRange(14 * 1024, 100));
 fileRanges.add(FileRange.createFileRange(10 * 1024, 100));
 fileRanges.add(FileRange.createFileRange(2 * 1024 - 101, 100));
 fileRanges.add(FileRange.createFileRange(40 * 1024, 1024));

 WeakReferencedElasticByteBufferPool pool = new WeakReferencedElasticByteBufferPool();
 ExecutorService dataProcessor = Executors.newFixedThreadPool(5);
 CountDownLatch countDown = new CountDownLatch(fileRanges.size());
```

```
try (FSDataInputStream in = fs.open(path(VECTORED_READ_FILE_NAME))) {
  in.readVectored(fileRanges, value -> pool.getBuffer(true, value));
  for (FileRange res : fileRanges)
    dataProcessor.submit(() -> {
      try {
        readBufferValidateDataAndReturnToPool(pool, res, countDown);
      } catch (Exception e) {
        LOG.error("Error while processing result for {} ", res, e);
      }});
  // user can perform other computations while waiting for IO.
  if (!countDown.await(100, TimeUnit.SECONDS))
    throw new AssertionError("Error while processing vectored io results");
 } finally {
  pool.release();
  HadoopExecutors.shutdown(dataProcessor, LOG, 100, TimeUnit.SECONDS);
}
}
```

```
private void readBufferValidateDataAndReturnToPool(
  ByteBufferPool pool, FileRange res, CountDownLatch countDownLatch)
       throws IOException, TimeoutException {

 CompletableFuture<ByteBuffer> data = res.getData();
 ByteBuffer buffer = FutureIO.awaitFuture(data,
         VECTORED_READ_OPERATION_TEST_TIMEOUT_SECONDS,
         TimeUnit.SECONDS);
 // Read the data and perform custom operation. Here we are just
 // validating it with original data.
 assertDatasetEquals((int) res.getOffset(), "vecRead",
         buffer, res.getLength(), DATASET);
 // return buffer to pool.
 pool.putBuffer(buffer);
 countDownLatch.countDown();
}
```

Is it faster? Oh yes*.

- JMH benchmark for local fs is 7x faster.

Hive queries with orc data in S3.

- TPCH => 10-20% reduced execution time.
- TPCDS => 20-40% reduced execution time.
- Synthetic ETL benchmark => 50-70% reduced execution time.

** your results may vary. No improvements for CSV*

JMH benchmarks

We initially implemented the new vectored IO API in RawLocal and Checksum filesystem and wrote JMH benchmark to compare the performance across

- Raw and Checksum file system.
- Direct and Heap buffers
- Vectored read vs traditional sync read api vs java async file channel read.

Data is generated locally and stored on local disk.

JMH benchmarks result

Benchmark	bufferKind	fileSystem	Mode	#	Score	Error	Units
VectoredReadBenchmark.asyncFileChanArray	direct	N/A	avgt	20	1448.505	± 140.251	us/op
VectoredReadBenchmark.asyncFileChanArray	array	N/A	avgt	20	705.741	± 10.574	us/op
VectoredReadBenchmark.asyncRead	**direct**	**checksum**	**avgt**	**20**	**1690.764**	**± 306.537**	**us/op**
VectoredReadBenchmark.asyncRead	direct	raw	avgt	20	1562.085	± 242.231	us/op
VectoredReadBenchmark.asyncRead	array	checksum	avgt	20	888.708	± 13.691	us/op
VectoredReadBenchmark.asyncRead	array	raw	avgt	20	810.721	± 13.284	us/op
VectoredReadBenchmark.syncRead	**(array)**	**checksum**	**avgt**	**20**	**12627.855**	**± 439.369**	**us/op**
VectoredReadBenchmark.syncRead	(array)	raw	avgt	20	1783.479	± 161.927	us/op

JMH benchmarks result observations.

- The current code (traditional read api) is by far the slowest and using the Java native async file channel is the fastest.
- Reading in raw fs is almost at par java native async file channel. This was expected as the vectored read implementation of raw fs uses java async channel to read data.
- For checksum fs vectored read is almost 7X faster than traditional sync read.

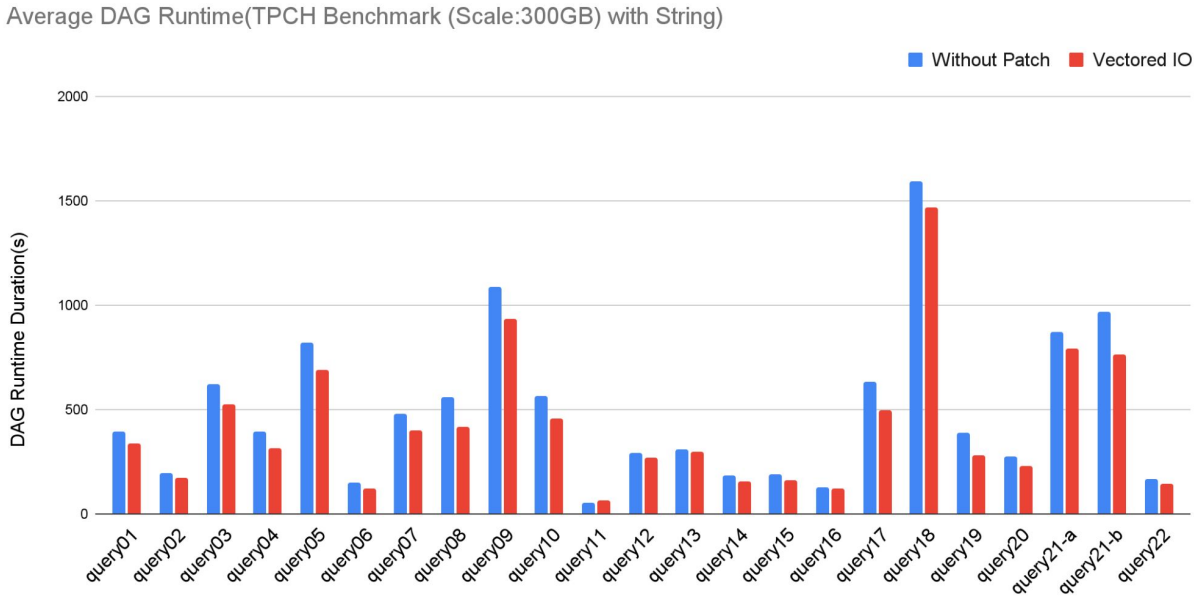
Hive with ORC data in S3:

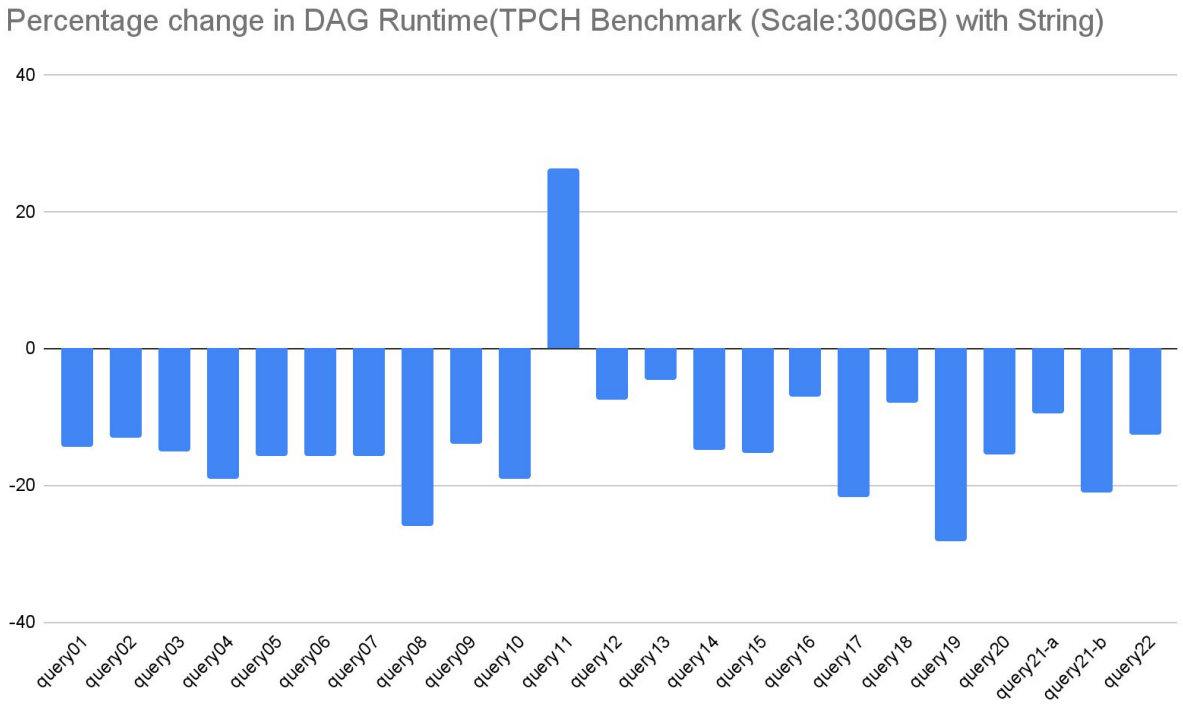
Modified ORC file format library to use the new API and ran hive TPCH, TPCDS and synthetic ETL benchmarks.

All applications using this format will automatically get the speedups. No changes are required explicitly.

Special mention to Owen O'Malley @linkedin who co-authored ORC and designed the initial vectored IO API.

Hive TPCH Benchmark:





NOTE: Q11 has a small runtime overall (just few secs) so can be neglected

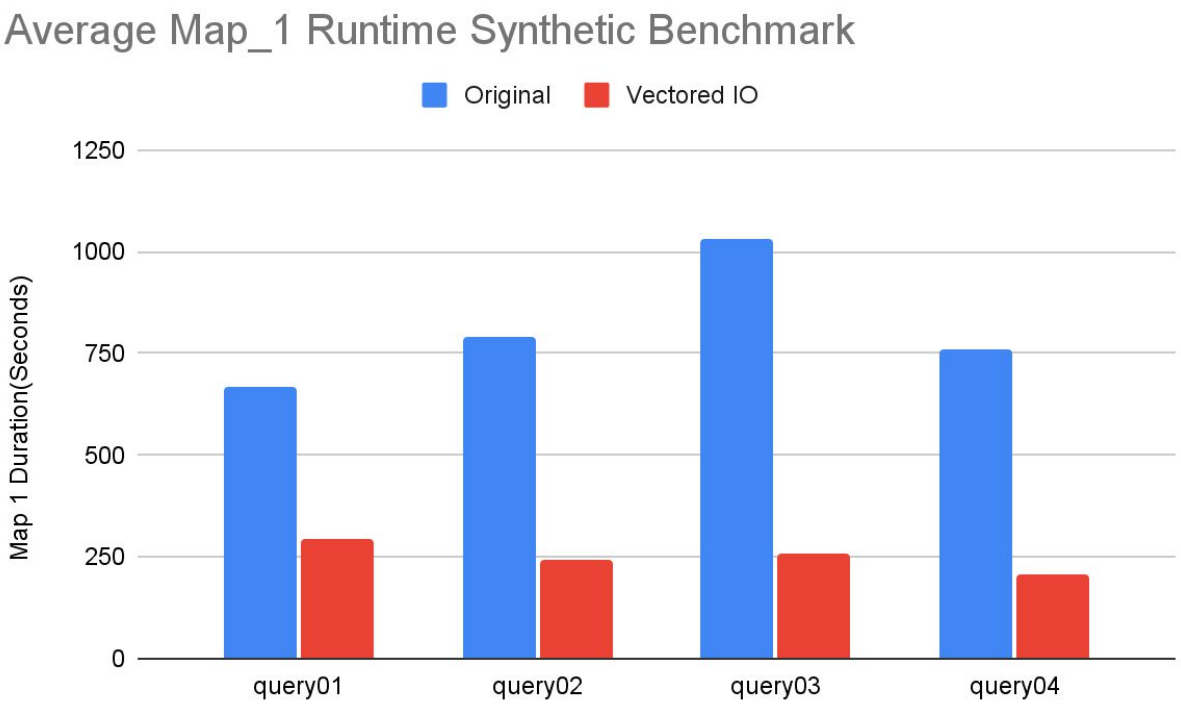
But the improvements were not super great.

TPCH benchmarks are too short to leverage full vectored io improvements.

We created a synthetic benchmark to test vectored IO and mimic ETL based scenarios.

In this benchmark we created wide string tables with large data reads.

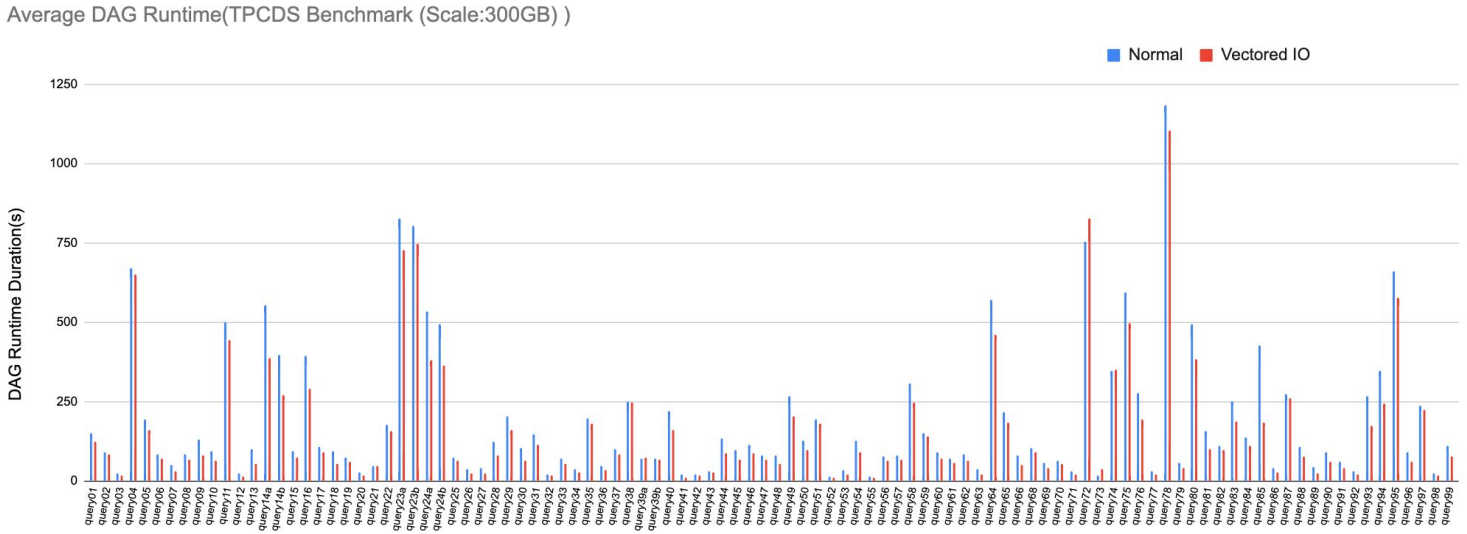
Queries that will generate up to 33 different ranges on the same stripe thus asking 33 ranges to be read in parallel.

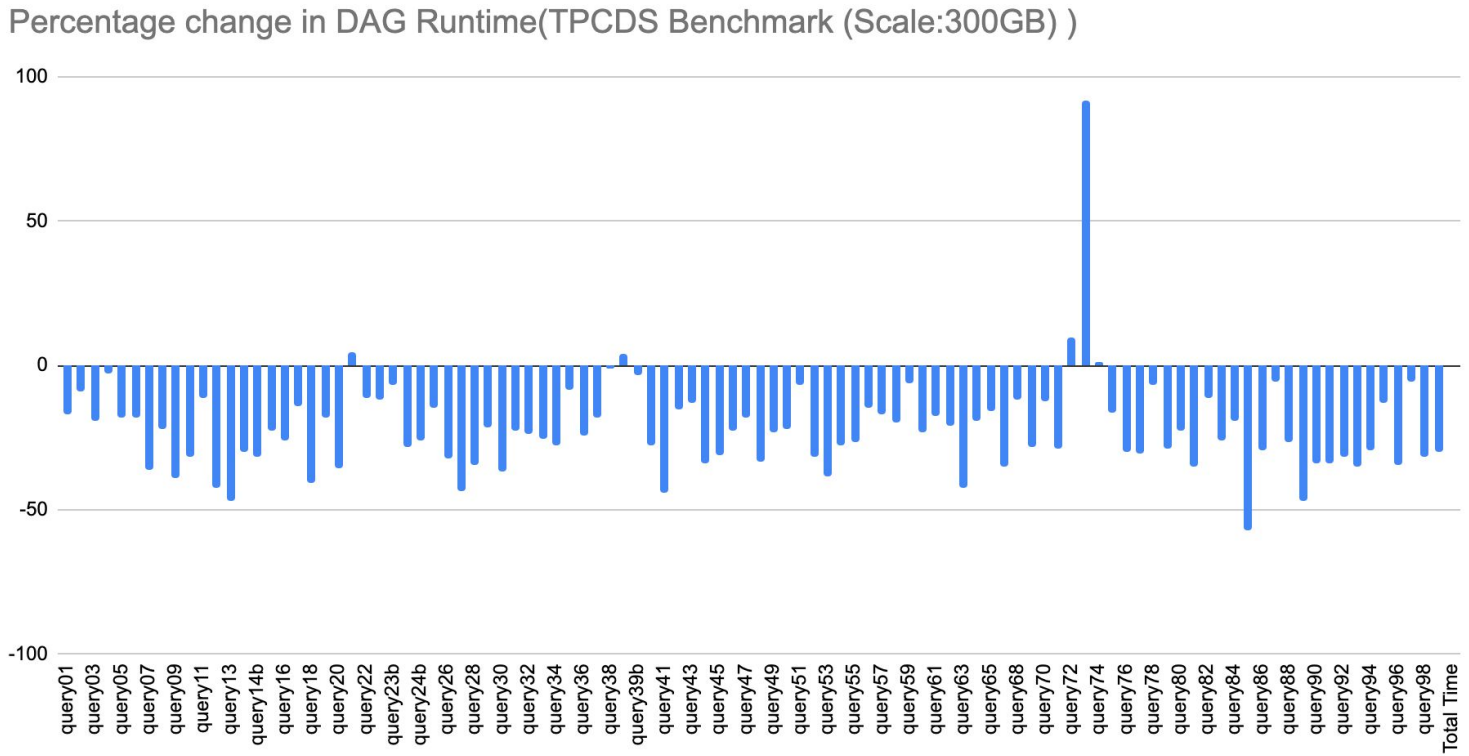

Average Map_1 Runtime Synthetic Benchmark
Original
Vectored IO
Map 1 Duration(Seconds)
1250
1000
750
500
250
0
query01
query02
query03
query04

Hive ETL benchmark result:

- 50-70 percent reduction in the mapper runtime with vectored IO.
- More performance gains in long running queries. This is because vectored IO fills the data in each buffer parallely.
- Queries which reads more data shows more speedup.

Hive TPCDS Benchmark





NOTE: Q74 has a small runtime overall (just few secs) so can be neglected for the sake of comparison. Tez container release time dominated.

Current State:

- API, default, local and S3A filesystem implementation in upcoming hadoop 3.3.5 release. HADOOP-18103

Upcoming work:

- ORC support in ORC-1251.
- Parquet support in PARQUET-2171. Running spark benchmarks.
- More hive benchmarks and figure out best default values for minSeek and maxReadSize.
- Perform load tests to understand more about memory and cpu consumption.
- ABFS and GCS implementation of Vectored IO.

Summary

- Vectored IO APIs can radically improve processing time of columnar data stored in cloud.
- ORC and Parquet libraries are the sole places we need to deliver this speedup
- API in forthcoming Hadoop release

Call to Action: Everyone

1. Upgrade to latest versions of Hadoop
2. And ORC, parquet libraries which use the new API
3. Enjoy the speedups!

Call to Action: Apache code contributors

- Make Hadoop 3.3.4 minimum release; ideally 3.3.5
- If you can't move to 3.3.5, help with
  HADOOP-18287. Provide a shim library for modern FS APIs
- Use IOStatistics APIs to print/collect/aggregate IO Statistics
- Use in your application.
- Extend API implementations to other stores.
- Test everywhere!

Thanks to everyone involved.

- Owen
- Steve
- Rajesh
- Harshit
- Shwetha
- Mehakmeet
- And many other folks from community.

Q/A?

Backup Slides

You said "Cheaper"?

- Range coalescing: fewer GET requests
- No wasted GETs on prefetch or back-to-back GETs
- Speedup in execution reduces life/cost of of VM clusters
- Or even reduces cluster size requirements

Spark support?

Comes automatically with Vector IO releases of ORC/Parquet and hadoop libraries

No changes to Spark needed at all!

tip: IOStatistics API gives stats on the operations; InputStream.toString() prints there.

Apache Avro

sequential file reading/processing ==> no benefit from use of the API

AVRO-3594: FsInput to use openFile() API

```
fileSystem.openFile(path)
  .opt("fs.option.openfile.read.policy", "adaptive")
  .withFileStatus(status)
  .build()
```

Eliminates HEAD on s3a/abfs open; read policy of "sequential unless we seek()".

(GCS support of `openFile()` enhancements straightforward...)

Apache Iceberg

- Iceberg manifest file IO independent of format of data processed
- If ORC/Parquet library uses Vector IO, query processing improvements
- As Iceberg improves query planning time (no directory scans), speedups as a percentage of query time *may* improve.
- Avro speedups (AVRO-3594, ...) do improve manifest read time.
- Even better: PR#4518: Provide mechanism to cache manifest file content

Apache Impala

Impala already has a high performance reader for HDFS, S3 and ABFS:

- Multiple input streams reading stripes on same file in different thread parallely.
- Uses unbuffer() to release all client side resources.
- Caches the existing unbuffered input streams for next workers.

HDFS Support

- Default `PositionedReadable` implementation works well as latency of `seek()/readFully()` not as bad as for cloud storage.
- Parallel block reads could deliver speedups if Datanode IPC supported this.
- Ozone makes a better target
- No active work here -yet

Azure and GCS storage

Azure storage through ABFS

- Async 2MB prefetching works OK for parquet column reads
- Parallel range read would reduce latency

Google GCS through gs://

- gs in random/adaptive does prefetch and cache footer already)
- Vectored IO adds parallell column reads
- `+openFile().withFileStatus`/seek policy for Avro and Iceberg.

Help needed here!

CSV

Just stop it!

Especially schema inference in Spark, which duplicates entire read

```
val csvDF = spark.read.format("csv")
  .option("inferSchema", "true")        /* please read my data twice! */
  .load("s3a://bucket/data/csvdata.csv")
```

Use Avro for exchange; ORC/Parquet for tables

HADOOP-18028. S3A input stream with prefetching

- PInterest code being incorporated into hadoop-trunk
- AWS S3 team leading this work
- Block prefetch; cache to memory or local FS
- Caches entire object into RAM if small enough (nice!)
- No changes to libraries for benefits
- Good for csv, avro, distcp, small files.

Stabilization needed for scale issues, especially in apps (Hive, Spark, Impala) with many threads

Will not deliver same targeted performance improvements as explicit range prefetching, but easier to adopt.

HADOOP-18287. Provide a shim library for FS APIs

- Library to allow access to the modern APIs for applications which still need to work with older 3.2+ Hadoop releases.
- Invoke new APIs on recent releases
- Downgrade to older APIs on older Hadoop releases

Applications which use this shim library will work on older releases but get the speedups on the new ones.

S3A configurations to tune:

```
<property>
  <name>fs.s3a.vectored.read.min.seek.size</name>
  <value>4K</value>
  <description>
     What is the smallest reasonable seek in bytes such
     that we group ranges together during vectored
     read operation.
   </description>
</property>
```

```
<property>
<name>fs.s3a.vectored.read.max.merged.size</name>
<value>1M</value>
<description>
   What is the largest merged read size in bytes such
   that we group ranges together during vectored read.
   Setting this value to 0 will disable merging of ranges.
</description>
</property>
```