Hadoop Vectored IO:
your data just got faster!

Steve Loughran

stevel@cloudera.com
github: steveloughran

CLOUDERA

Steve Loughran

- Long-standing open source developer
- Hadoop Committer
- Cloud Connectors (s3a, abfs…)
- PB of data goes through my code every day
- I get to care when it doesn't or just takes too long!
- Mountain Biking (+climbing)

CLOUDERA

Problem: reading large data from cloud storage is slow

- More time waiting for queries.
- Longer cluster lifespans or more VMs
- Bigger bills from AWS, Microsoft, Google.
- Worst case: ETL jobs take so long they can't keep up

Vectored IO transforms ORC and Parquet read times in cloud

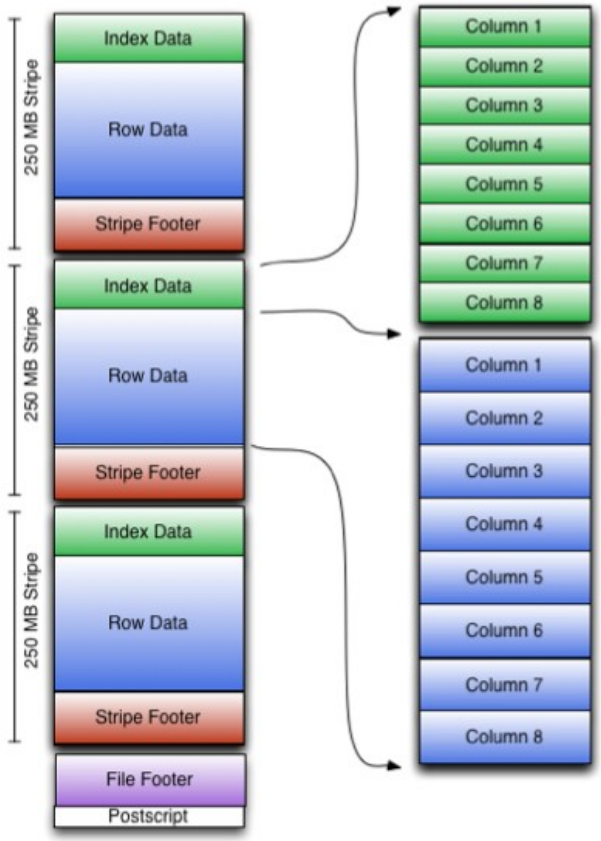
`seek()` is a concept from Hard Disks: its time is done

- HDDs: seeking *was* a physical operation
- SSDs: block-by-block reads; *you can read > 1 block in parallel*.
- Cloud storage: S3, Azure, GCS: latency of GET requests
- Prefetching? Great for *sequential* IO
- ORC/Parquet reads are considered "random IO", but there is nothing random about it

If the format readers can pass their "read plan" to the filesystem client, it can optimise data retrieval

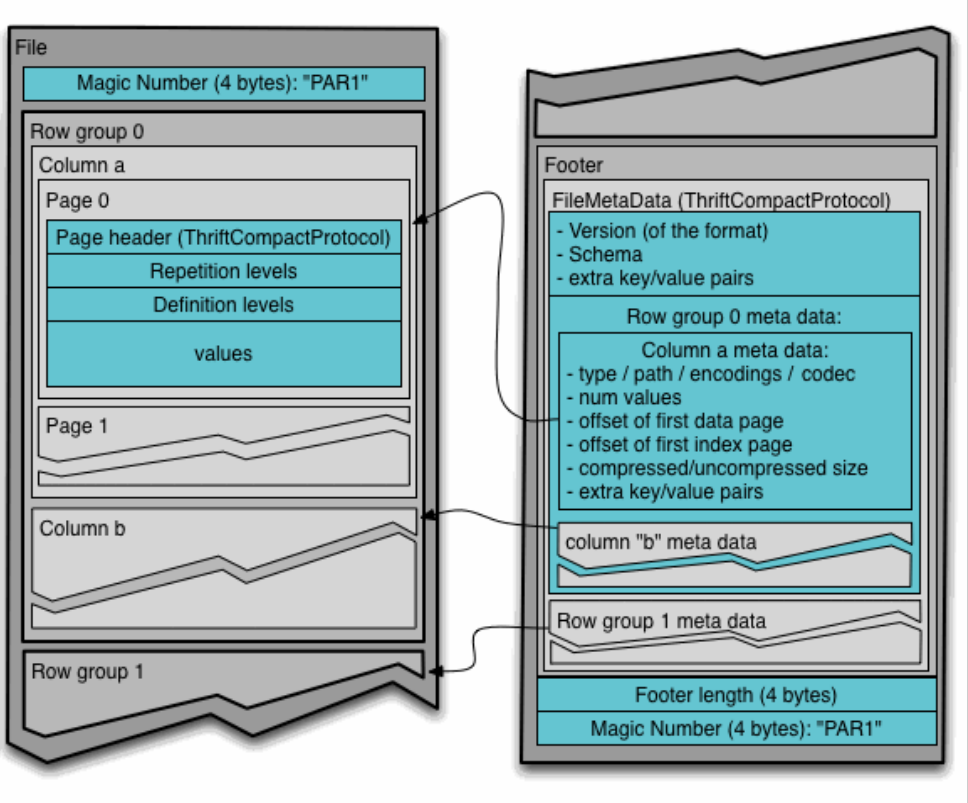
What do you mean "random" IO?

ORC



1. Fixed length postscript bytes contain offset of footer
2. Footer includes schemas, stripe info
3. Index data in stripes include column ranges, encryption, compression
4. Column stripes of hundreds of MB
5. Columns can be read with this data
6. Predicate pushdown can dynamically skip columns if index data shows it is not needed.

Parquet



- Thrift-based columnar format
- 256+ MB Row Groups
- Fixed 8 byte postscript includes pointer to real footer
- Footer has schema + metadata of every column in every row group
- Query engines process subset of columns within row groups

CLOUDERA

Columnar formats need Hadoop Vectored IO

- Inspired from `readv()/writev()` from Linux.
- Scatter/Gather API where callers request 1+ file range.
- Each range mapped to a `CompletableFuture`
- Default implementation: sort ranges + sequential `readFully(offset, range)` into buffers
- Object store optimizations.

Works great everywhere; radical benefit in object stores

New method in `PositionedReadable` interface:

```
public interface PositionedReadable {

 readVectored(
   List<? extends FileRange> ranges,
   IntFunction<ByteBuffer> allocator)
     throws IOException

}
```

default implementation: available everywhere

Advantages

Asynchronous: 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

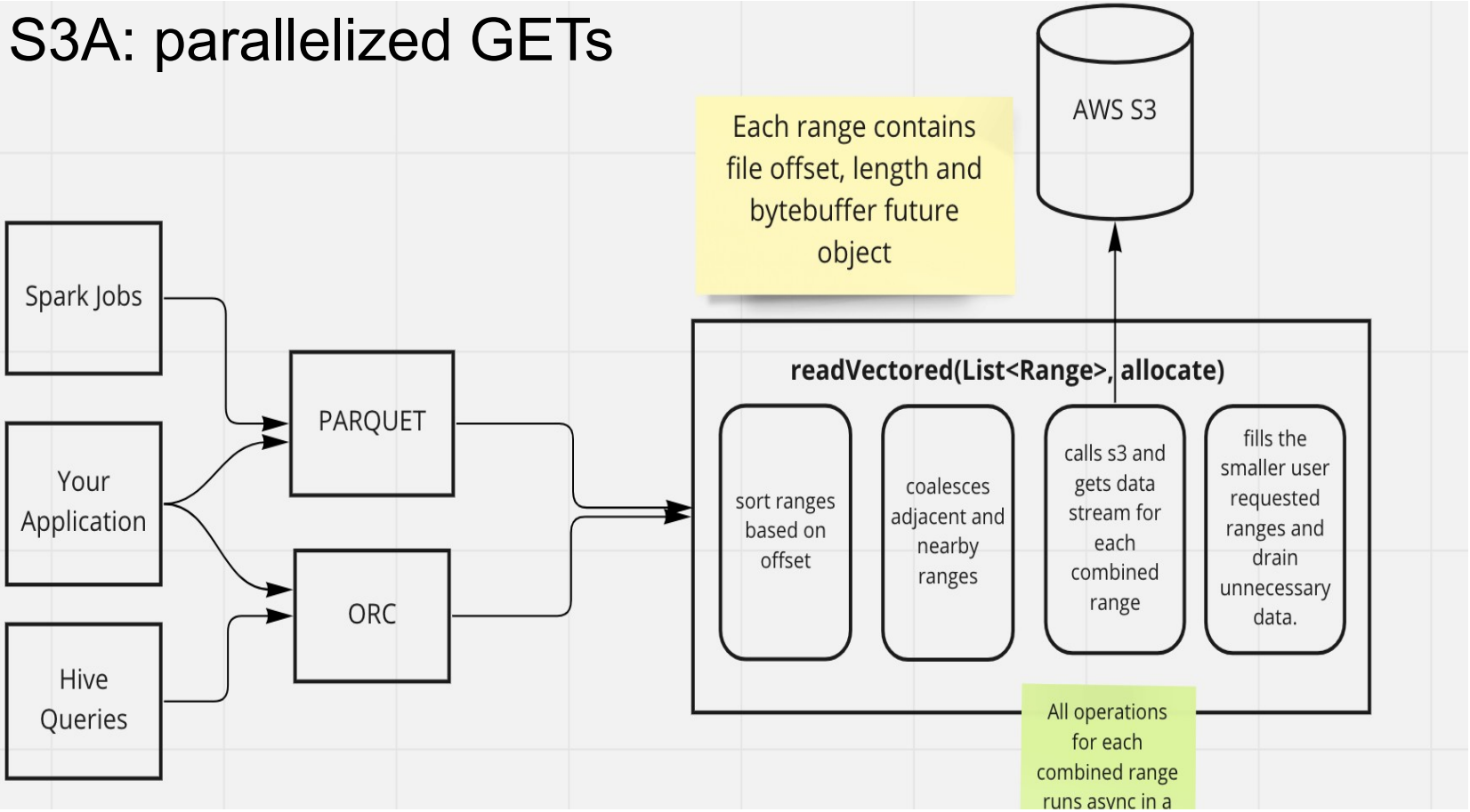
Default implementation:

- Blocking `readFully()` calls for each range.
- Same performance as the classic API.

Local/Checksum FileSystem:

- Java NIO Async Channel API
- Reads directly into on-heap or off-heap ByteBuffers
- Parallel read with zero-kernel-copy, DMA, memory-mapped SSD...

S3A: parallelized GETs



S3A parallel GET requests

- Multiple GET requests issued into common thread pool.
- Set in `fs.s3a.vectored.active.ranged.reads`; default = 4
- Results returned as soon as their data is received
- If/when S3 supports multiple ranges in GET, could be done in a single request.

How to use?

```
Path p = path("s3a://bucket/table1/file.orc");
FileSystem fs = FileSystem(p.getUri(), conf);

List<FileRange> ranges = new ArrayList<>();
ranges.add(createFileRange(8 * 1024, 100));
ranges.add(createFileRange(14 * 1024, 100));
ranges.add(createFileRange(10 * 1024, 100));
ranges.add(createFileRange(2 * 1024 - 101, 100));
ranges.add(createFileRange(40 * 1024, 1024));

ByteBufferPool pool = new ElasticByteBufferPool();
ExecutorService dataProcessor = Executors.newFixedThreadPool(5);
```

```
// open file
try (FSDataInputStream in = fs.open(p)) {
  // initiate vectored read
  in.readVectored(ranges, size -> pool.getBuffer(false, size));

  // process results in worker pool
  for (FileRange range : ranges)
    dataProcessor.submit(() ->
      range.getData().thenAccept(buffer ->  {
          /* process */
        pool.putBuffer(buffer);
        countDownLatch.countDown();
    }));
  countDown.await();
}
```

CLOUDERA

Is it faster? Oh yes*.

Java Microbench benchmark for local storage 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 or Avro*

Java Microbench Harness

Native IO implementation of vectored IO API in RawLocal and Checksum filesystem

JMH benchmark to compare the performance across

- FS: [Raw, Checksum file system].
- Buffers: [Direct, Heap]
- read: [`seek()` + `read()`, Vectored, java async file channel read]

see: `org.apache.hadoop.benchmark.VectoredReadBenchmark`

CLOUDERA

JMH benchmarks result

Benchmark	buffer	fileSystem	Score	Error
syncRead	**(array)**	**checksum**	**12627.855**	**± 439.369**
syncRead	(array)	raw	1783.479	± 161.927
asyncFileChanArray	direct	N/A	1448.505	± 140.251
asyncFileChanArray	**array**	**N/A**	**705.741**	**± 10.574**
asyncRead	direct	checksum	1690.764	± 306.537
asyncRead	direct	raw	1562.085	± 242.231
asyncRead	**array**	**checksum**	**888.708**	**± 13.691**
asyncRead	array	raw	810.721	± 13.284

=7x faster

(microseconds per operation)

CLOUDERA

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.

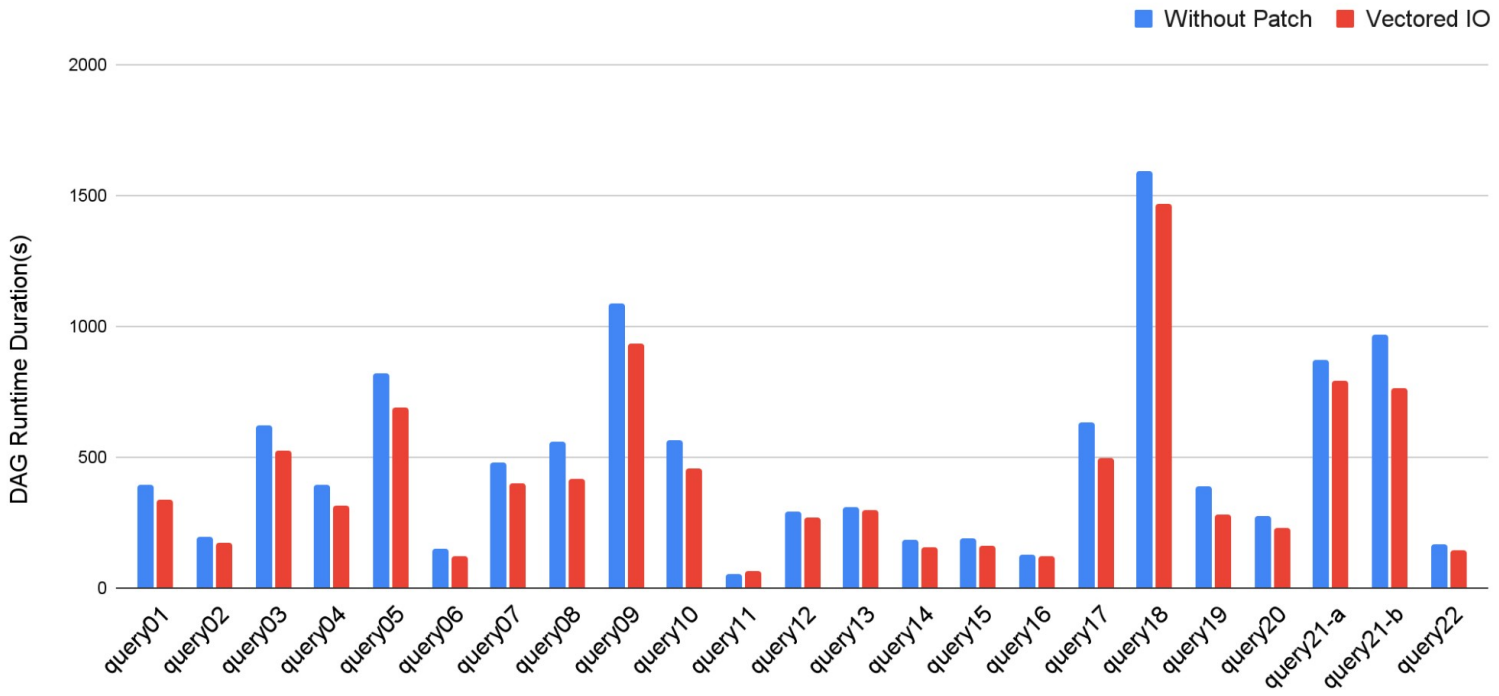
Hive with ORC data in S3:

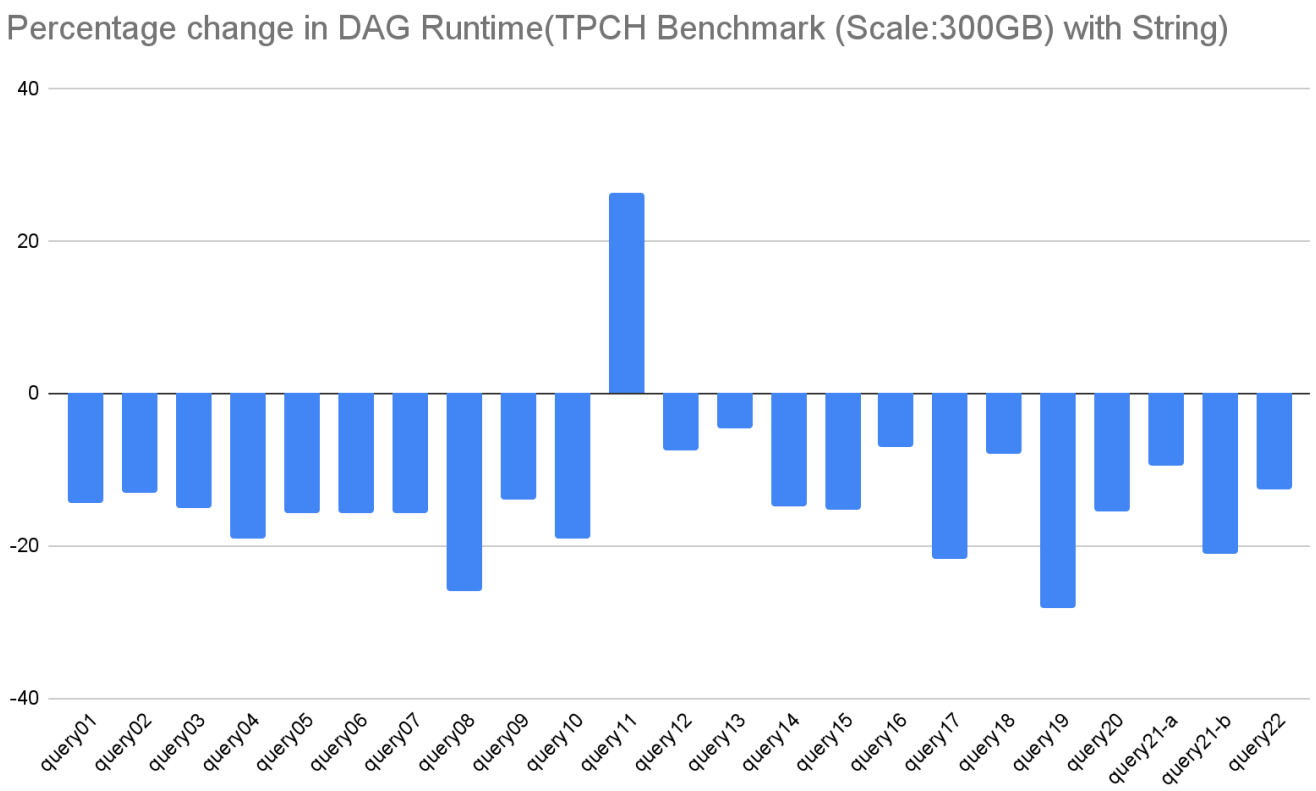
ORC library updated to use to the new API when reading stripes

All applications using this format automatically get the speedups.

Special mention to Owen O'Malley

Hive TPCH Benchmark:
10-20% decrease in query runtimes

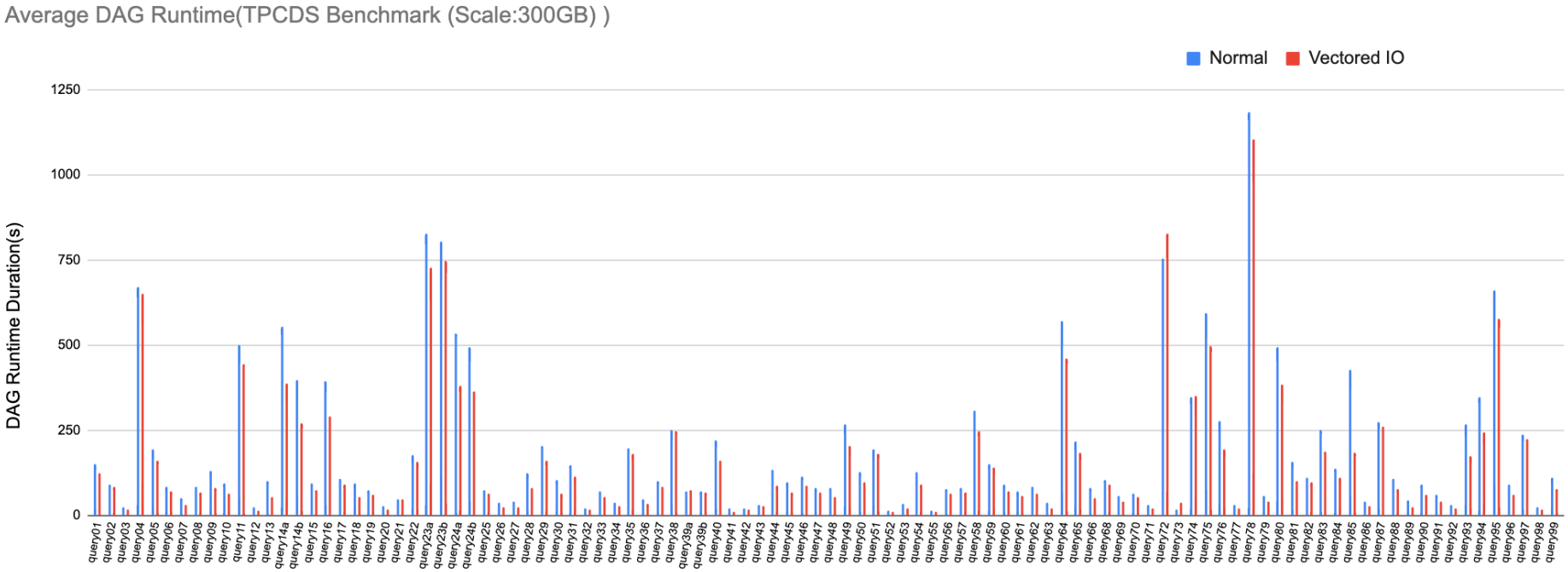




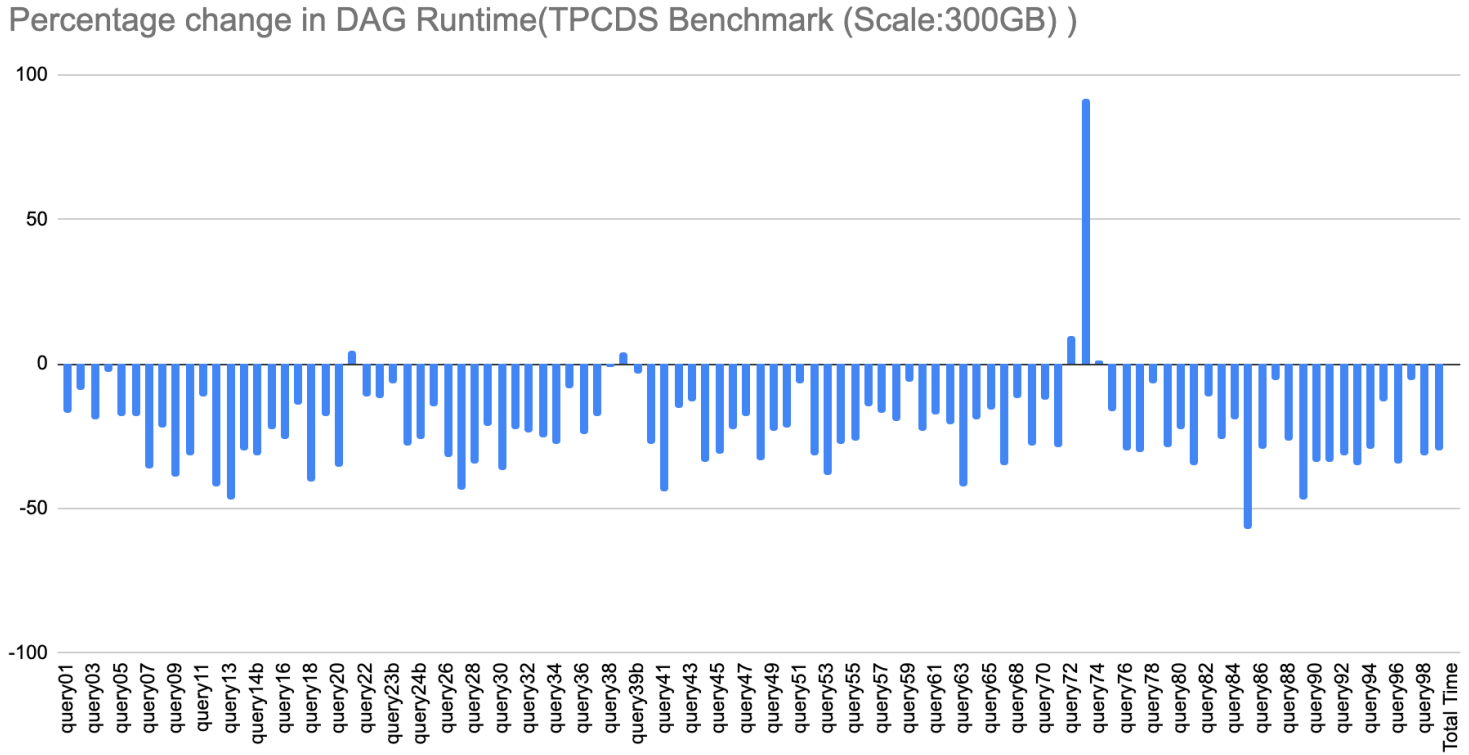
NOTE: Q11 has a small runtime overall (just few secs) so can be neglected

CLOUDERA

Hive TPCDS Benchmark @ 300GB



CLOUDERA



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

CLOUDERA

Synthetic benchmark of ETL & vectored IO

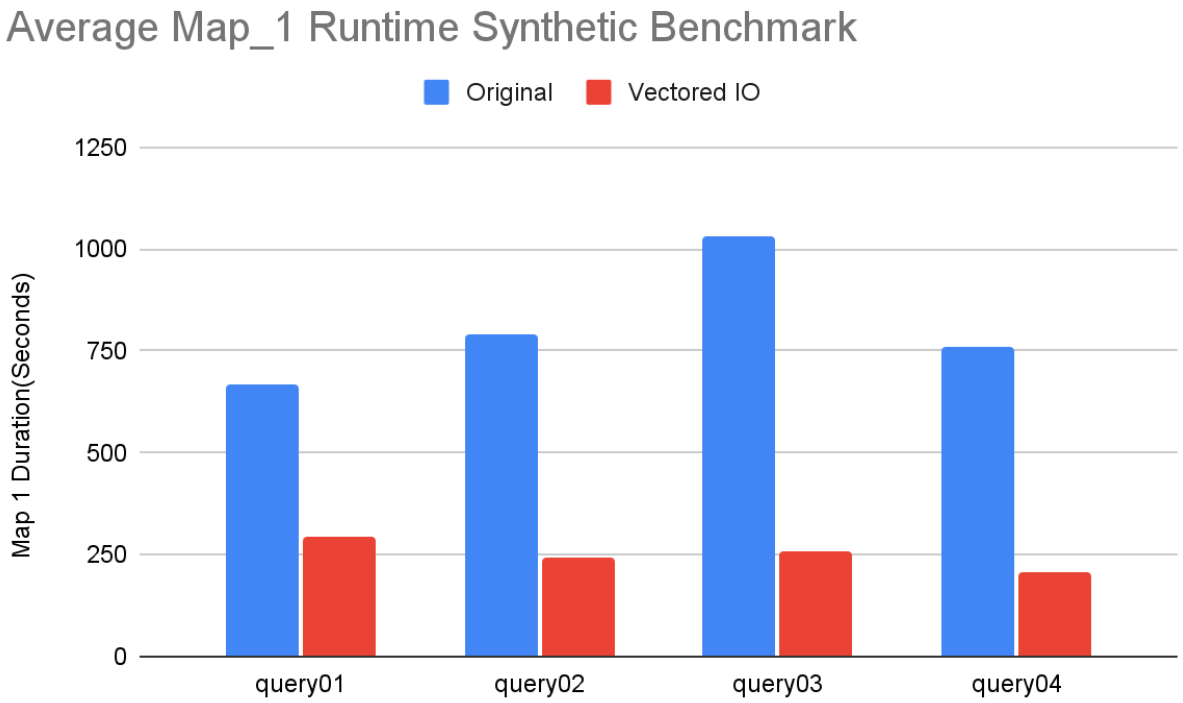
TPCH benchmarks are too short to show full vectored IO improvements.

Synthetic benchmark to test vectored IO and mimic ETL based scenarios.

Wide string tables with large data reads.

Queries generate up to 33 different ranges on the same stripe

Synthetic benchmark of ETL & vectored IO



CLOUDERA

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 in parallel.
- Queries which reads more data shows more speedup.

Shipping in Hadoop 3.3.5!

- Available in Hadoop 3.3.5
- file:// has local filesystem NIO implementation
- S3A does parallel GETs per input stream
- Fallback implementation for everything else

Azure and GCS storage: help needed

Azure storage through ABFS

- Async 2MB prefetching doesn't do much for columnar formats
- Parallel range read would reduce latency

Google GCS through gs://

- random/adaptive policies do prefetch and cache footer already
- Vectored IO adds parallel column reads
- `+openFile()` supports status and seek policy for Avro and Iceberg.

Problem: the format libraries are stuck

Update the format libraries & the apps get the speedup *automatically*

- **ORC-1251**. Support VectoredIO in ORC
- **PARQUET-2171.** Implement vectored IO in parquet file format
- +Iceberg bundles its own shaded Parquet lib

How to get this into libraries that still want to support Hadoop 2.x?

1. Cloudera: we can ship our own consistent set of libraries.
2. WiP: https://github.com/apache/hadoop-api-shim
3. *For now: apply our patches yourself*

Conclusions

- Vectored IO APIs radically improve processing time of columnar data stored in cloud and local SSD.
- Available in Hadoop 3.3.5
- Azure and GCS connectors next targets
- We need to get support into ORC and Parquet libraries
- We need to work together to get support into the ASF releases

Questions?

Backup Slides

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 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...)

Spark support?

Comes automatically with Vector IO releases of ORC/Parquet and Iceberg dependencies

Spark master branch on Hadoop 3.3.5, so ready to play

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.
- 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

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-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.