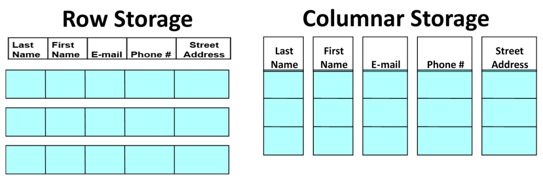
DBMS Musings

Tuesday, October 31, 2017

Apache Arrow vs. Parquet and ORC: Do we really need a third Apache project for columnar data representation?

Apache Parquet and Apache ORC have become a popular file formats for storing data in the Hadoop ecosystem. Their primary value proposition revolves around their “columnar data representation format”. To quickly explain what this means: many people model their data in a set of two dimensional tables where each row corresponds to an entity, and each column an attribute about that entity. However, storage is one-dimensional --- you can only read data sequentially from memory or disk in one dimension. Therefore, there are two primary options for storing tables on storage: Store one row sequentially, followed by the next row, and then the next one, etc; or store the first column sequentially, followed by the next column, and then the next one, etc.



Storage layout difference between row- and column-oriented formats

For decades, the vast majority of data engines used row-oriented storage formats. This is because many early data application workloads revolved around reading, writing, and updating single entities at a time. If you store data using a columnar format and you want to extract all attributes for a particular entity, the system must jump around, finding the data for that entity from each of the separately-stored attributes. This results in a random access pattern, which results in slower access times than sequential access patterns. Therefore, columnar storage formats are a poor fit for workloads that tend to read and write entire entities at once, such as OLTP (transactional) workloads. Over time, workloads became more complex, and data analytics workloads emerged that tended to focus on only a few attributes at once; scanning through large quantities of entities to aggregate and/or process values of these attributes. Thus, storing data in a columnar fashion became more viable, and columnar formats resulted in sequential, high performance access patterns for these workloads.

Apache Arrow has recently been released with seemingly an identical value proposition as Apache Parquet and Apache ORC: it is a columnar data representation format that accelerates data analytics workloads. Yes, it is true that Parquet and ORC are designed to be used for storage on disk and Arrow is designed to be used for storage in memory. But disk and memory share the fundamental similarity that sequential access is faster than random access, and therefore the analytics workloads which tend to scan through attributes of data will perform more optimally if data is stored in columnar format no matter where data is stored --- in memory or on disk. And if that’s the case, the workloads for which Parquet and ORC are a good fit will be the identical set of workloads for which Arrow is a good fit. If so, why do we need two different Apache projects?

Before we answer this question, let us run a simple experiment to validate the claimed advantages of column-stores. On an Amazon EC2 t2.medium instance, I created a table with 60,000,000 rows (entities) in main memory. Each row contained six attributes (columns), all of them 32-bit integers. Each row is therefore 24 bytes, and the entire table is almost 1.5GB. I created both row-oriented and column-oriented versions of this table, where the row-oriented version stores the first 24-byte row, followed by the next one, etc; and the column-oriented version stores the entire first column, followed by the next one, etc. I then ran a simple query ideally suited for column-stores --- I simply search the entire first column for a particular value. The column-oriented version of my table should therefore have to scan though just the first column and will never need to access the other five columns. Therefore it will need to scan through 60,000,000 values * 4 bytes per value = almost 0.25GB. Meanwhile the row-store will need to scan through the entire 1.5GB table because the granularity with which data can be passed from memory to the CPU (a cache line) is larger than the 24-byte tuple. Therefore, it is impossible to read just the relevant first attribute from memory without reading the other five attributes as well. So if the column-store has to read 0.25GB of data and the row-store has to read 1.5GB of data, you might expect the column-store to be 6 times faster than the row-store. However, the actual results are presented in the table below:

Daniel Abadi

About Me

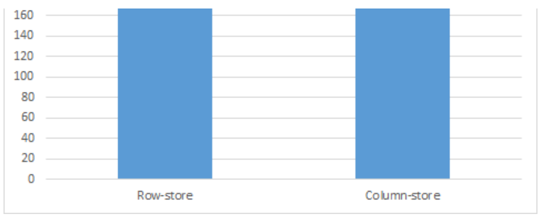
Daniel Abadi

Daniel Abadi is the Darnel
Computer Science at Univ
Park. He is best-known fo
storage and query executi
(column-oriented database
commercialized by Vertica
Hewlett-Packard in 2011, f
on fault tolerant scalable a
which was commercialized
Teradata in 2014, and dete
transactional, distributed s
which is currently being co
Abadi has been a recipien
a NSF CAREER Award, a
a VLDB Best Paper Award
Paper Awards, the 2008 S
Dissertation Award, the 20
Teaching Prize, and the 20
Researcher Award. He rec
MIT. He blogs at DBMS M
@daniel_abadi.

View my complete profile

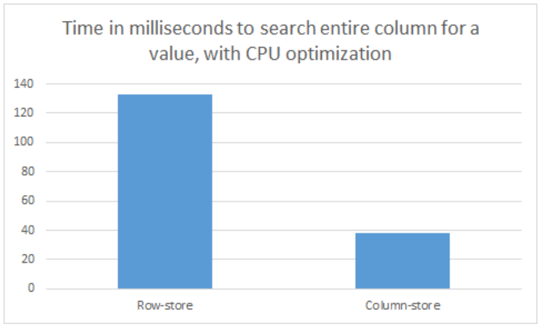
Blog Archive

- ► 2019 (8)
- ► 2018 (3)
- ▼ 2017 (3)
  - ▼ October (2)
    - Apache Arrow vs. Parquet and ORC: Do we really nee...
    - Hazelcast and the Mythical PA/EC System
  - ► April (1)
- ► 2015 (1)
- ► 2012 (4)
- ► 2011 (6)
- ► 2010 (8)
- ► 2009 (21)



Surprisingly, the row-store and the column-store perform almost identically, despite the query being ideally suited for a column-store. The reason why this is the case is that I turned off all CPU optimizations (such as vectorization / SIMD processing) for this query. This resulted in the query being bottlenecked by CPU processing, despite the tiny amount of CPU work that has to happen for this query (just a simple integer comparison operation per row). To understand how it is possible for such a simple query to be bottlenecked by the CPU, we need to understand some basic performance specifications of typical machines. As a rough rule of thumb, sequential scans through memory can feed data from memory to the CPU at a rate of around 30GB a second. However, a modern CPU processor runs at approximately 3 GHz --- in other words they can process around 3 billion instructions a second. So even if the processor is doing a 4-byte integer comparison every single cycle, it is processing no more than 12GB a second --- a far smaller rate than the 30GB a second of data that can be sent to it. Therefore, CPU is the bottleneck, and it does not matter that the column-store only needs to send one sixth of the data from memory to the CPU relative to a row-store.

On the other hand, if I turn on CPU optimizations (by adding the '-O3' compiler flag), the equation is very different. Most notably, the compiler can vectorize simple operations such as the comparison operation from our query. What this means is that originally each of the 60,000,000 integer comparisons that are required for this query happened sequentially --- each one occurring in a separate instruction from the previous. However, once vectorization is turned on, most processors can actually take four (or more) contiguous elements of our column, and do the comparison operation for all four of these elements in parallel --- in a single instruction. This effectively makes the CPU go 4 times faster (or more if it can do more than 4 elements in parallel). However, this vectorization optimization only works if each of the four elements fit in the processor register at once, which roughly means that they have to be contiguous in memory. Therefore, the column-store is able to take advantage of vectorization, while the row-store is not. Thus, when I run the same query with CPU optimizations turned on, I get the following result:



As can be seen, the EC2 processor appears to be vectorising 4 values per instruction, and therefore the column-store is 4 times faster than the row-store. However, the CPU still appears to be the bottleneck (if memory was the bottleneck, we would expect the column-store to be 6 times faster than the row-store).

We can thus conclude from this experiment that column-stores are still better than row-stores for attribute-limited, sequential scan queries like the one in our example and similar queries typically found in data analytics workloads. So indeed, it does not matter whether the data is stored on disk or in memory --- column-stores are a win for these types of workloads. However, the reason is totally different. When the table is stored on disk, the CPU is much faster than the bandwidth with which data can be transferred from disk to the CPU. Therefore, column-stores are a win because they require less data to be transferred for these workloads. On the other hand, when the table is stored in memory, the amount of data that needs to be transferred is less relevant. Instead, column-stores are a win because they are better suited to vectorized processing.

The reason why it is so important to understand the difference in bottleneck (even though the bottom line is the same) is that certain decisions for how to organize data into storage formats will look different depending on the bottleneck. Most notably, compression decisions will look very different. In particular, for data stored on disk, where the bandwidth of getting data from disk to CPU is the bottleneck, compression is almost always a good idea. When you compress data, the total size of the data is decreased, and therefore less data needs to be transferred. However, you may have to pay additional CPU processing costs to do the decompression upon arrival. But if CPU is not the bottleneck, this is a great tradeoff to make. On the other hand, if CPU is the bottleneck, such as our experiments above where the data was located in main memory, the additional CPU cost of decompression is only going to slow down the query.

for memory-resident-data, does not support these algorithms. The only compression currently supported by Arrow is dictionary compression, a scheme that usually does not require decompression before data processing. For example, if you want to find a particular value in a data set, you can simply search for the associated dictionary-encoded value instead. I assume that the Arrow developers will eventually read my 2006 paper on compression in column-stores and expand their compression options to include other schemes which can be operated on directly (such as run-length-encoding and bit-vector compression). I also expect that they will read the X100 compression paper which includes schemes which can be decompressed using vectorized processing. Thus, I expect that Arrow will eventually support an expanded set of compression options beyond just dictionary compression. But it is far less likely that we will see heavier-weight schemes like gzip and snappy in the Apache Arrow library any time soon.

Another difference between optimizing for main memory and optimizing for disk is that the relative difference between random reads and sequential reads is much smaller for memory than for disk. For magnetic disk, a sequential read can be 2-3 orders of magnitude faster than a random read. However, for memory, the difference is usually less than an order of magnitude. In other words, it might take hundreds or even thousands of sequential reads on disk in order to amortize the cost of the original random read to get to the beginning of the sequence. But for main memory, it takes less than ten sequential reads to amortize the cost of the original random read. This enables the batch size of Apache Arrow data to be much smaller than batch sizes of disk-oriented storage formats. Apache Arrow actually fixes batches to be no more 64K records.

So to return back to the original question: do we really need a third column-store Apache project? I would say that there are fundamental differences between main-memory column-stores and disk-resident column-stores. Main-memory column-stores, like Arrow, need to be CPU optimized and focused on vectorized processing (Arrow aligns data to 64-byte boundaries for this reason) and low-overhead compression algorithms. Disk-resident column-stores need to be focused transfer-bandwidth and support higher-compression ratio algorithms. Therefore, it makes sense to keep Arrow and Parquet/ORC as separate projects, while also continuing to maintain tight integration.

Posted by Daniel Abadi at 8:49 AM

16 comments:

Geoff Langdale November 1, 2017 at 1:07 PM

Is the fact that your benchmark example isn't compute bound due to the instance not necessarily being capable of using a modern ISA? I would hope to be able to get 8-16 32-bit integer compares a cycle with AVX2 or AVX512 in the steady state (memory permitting; also branch misses permitting assuming we take a branch when we see something).

Reply

Replies

Daniel Abadi November 1, 2017 at 1:15 PM

Hi Geoff,

What I was trying to say is that AVX2 or AVX512 is far less helpful for row-stores than column-stores, since in row-stores the register is polluted with data that will not be operated on.

Geoff Langdale November 1, 2017 at 3:50 PM

Yes, indeed, and you showed admirable restraint not making the example even more stark (it could have been 16 columns not 6, and the column of interest could have been 6 bits wide not 32) :-)

Reply

Unknown November 8, 2017 at 1:13 AM

Nice article!

but just a simple question: Why you give the memory bandwidth from memory to CPU is 30GB. The DDR3-2133 is about 18.3 GB/s, DDR4-3200 is just about 25.6 GB/s. The number you give exaggerate the CPU process rate and memory bandwidth. What's more, there are lots of CPU cores in one machine, you just do not mention this enough.

Reply

Replies

Daniel Abadi November 8, 2017 at 3:24 PM

I said "around 30GB". I was just trying to give ballpark numbers rather than exact figures.

As far as the number of CPU cores --- this post should be understood on a per-core basis. I agree that if there are many cores, all

Unknown November 8, 2017 at 5:40 PM

Thanks for your reply!

I agree with your conclusion. Just the number you given can lead to the evidence is not so strong:-)

I consider the same problem for a while, Do we really need the Apache Arrow project ? Thanks for this article!

Reply

Unknown November 8, 2017 at 5:48 PM

Hope you can give more think about how to combine the Disk-based format and the Memory-based format.

From my experience, customers may want to cache data in memory, but the cost is so high, and want to put data back to disk if the resource is not enough. If one format can be flexible to support this, it maybe a good choose for users.

Reply

Anonymous January 3, 2018 at 2:05 AM

This comment has been removed by the author.

Reply

Anonymous January 3, 2018 at 2:38 AM

Good article but there are some important points missed here.

The most important being that row-oriented storage also implies one other thing: in all real world implementations row data is not packed contiguously like an array of C structs which this post implies (in absence of source code). That's because it has to allow updates/deletes to rows. Typical in-memory stores will use some form of hash map for in-memory cache. An interesting aside is that linked hash maps typically fare better in scans than normal or even open hash maps when row sizes are bit large (like >100 bytes).

Secondly row format allowing for varying column types also means that it has to maintain those offsets somewhere in schema and read those offsets while decoding thus additional reads are required breaking the cache lines further. On the other hand column format for primitive types have fixed sizes (or ones that can be read/jumped over in batches like run-length or efficient encodings like FastPFOR which are not possible with row format).

All in all, raw speeds per core for a single integer/long in rows with 24 bytes sizes is ~50ns per row per core and typically more, and cannot become much better no matter what one does. Compared to that column formats can do <1ns per row with/without encodings as the post also shows.

Some other points worth noting in the article and comments:

"So even if the processor is doing a 4-byte integer comparison every single cycle, it is processing no more than 12GB a second"

Superscalar architecture can do like 4 instructions per cycle even without SIMD. Not sure about EC2 but on my laptop typically get <1ns per integer/long (per core) for even average processing (which are more instructions than simple equals comparison). Adding additional filters on the same integer/long column hardly changes the numbers.

"But it is far less likely that we will see heavier-weight schemes like gzip and snappy in the Apache Arrow library any time soon"

Don't know about Apache Arrow, but compressing in memory means one can fit lot more in memory. Consider typical speed of decompression with schemes like LZ4 which is of the range of ~2GB/s compared to ~500MB/s disk reads even with SSDs, its a win in most scenarios (assuming no spare RAM for OS buffers).

"However, for memory, the difference is usually less than an order of magnitude."

Not true. Typical numbers are like ~4 cycles for L1, ~10 for L2, ~40 for L3 and ~100 or more for RAM. So the relative difference of sequential vs random is similar whether its disk or memory. Besides all parquet/ORC scanners will do sequential column block reads as far as possible, skipping forward in the same file as required.

"On the other hand the amount of processing done per data item is tiny in this example; in a real system there is generally much more CPU overhead per data item."

In typical queries, problem still remains the RAM/cache to CPU speed. For example even simple joins with reference data that fits in L1/L2 cache, the best hash joins will have to jump on the hash buckets and are typically an order of magnitude slower. Adding more filters, especially on primitives, hardly effects the numbers. For bigger joins the hits are even more (and of course, if a shuffle/sort gets introduced then those will completely dominate the numbers). The case where CPU matters are simple full scan queries with multiple complex filters but those are serviced much better using indexes.

file gets cached in OS buffers. The memory optimized engine we have built at SnappyData can go about 5-10X faster in the best cases, but less for complex queries. The bigger advantages lie elsewhere like indexing, minimizing shuffle, hybrid store etc.

Reply

Replies

Daniel Abadi January 8, 2018 at 4:18 PM

Thanks for the comment and the more precise numbers in several places. I do disagree about a couple of things though:

(1) Updatability is mostly orthogonal to row vs. column. Both can dense-pack data if updates are not allowed, and both are more likely to use less read-optimal data layouts if updates are allowed.

(2) Real world DB execution engines are more CPU intensive than what you indicate. (At least in my experience)

Anonymous January 10, 2018 at 8:53 AM

Thanks for the reply.

1) Not necessarily for column format, especially if the update layout is also using column format. Our experience has been that one can get do with 2-3X degradation with even large/frequent updates.

2) Will defer to your experience.

Unknown October 22, 2018 at 5:24 AM

Very interesting article and debate. Maybe using Parquet/ORC in a RAM disk, taking into account the copying overhead of course, could contribute some more numbers to your debate.

Reply

RD March 28, 2018 at 6:27 PM

Really nice article!

Reply

durga prasad kumar May 22, 2019 at 1:14 PM

Really good article ... thanks you

Reply

Mobbcore July 31, 2019 at 6:31 PM

"For decades, the vast majority of data engines used row-oriented storage formats"

Well, that's not quite true. While speaking about decades and rows vs columns, APL, A+, J, K, Q, kdb+ should definitely be mentioned. Beloved pets of stock exchanges and power plant engineers for decades. Essentially columnar.

Reply

Replies

Daniel Abadi August 1, 2019 at 10:21 AM

Thanks for the comment.

Like I said ... "vast majority". You bring examples of column-stores. Sybase IQ is another historical example. But the vast majority of data engines were row-stores :)

Reply

Add comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)