Hashing out Perfect Group-Bys in a GPU-accelerated database

Todd Mostak
February 2, 2018

One of the fundamental operations in SQL is grouping by a set of columns (often known as dimensions) and rolling up aggregates (i.e. COUNT, SUM, AVG…) for another set (often known as measures. Since MapD was built first-and-foremost for high performance analytical processing across both CPU and GPU, it was critical to make such queries execute as fast possible.

Since MapD can leverage both the CPU and GPU to process SQL, it was also important to deploy query execution strategies that were tailored to the strengths of each architecture. For the most part, what is optimal for CPU is optimal for the GPU, but there are some unique considerations when executing on GPU that had to be accounted for, as outlined below.

Fundamental to understanding the expected performance of different strategies is to understand how a processor caches data. Accessing the L1 and L2 caches of both CPUs and GPUs both exhibit significantly higher bandwidth and lower latency than accesses to the main memory. Thus to achieve best performance it is critical that both reads and writes in group by and other operations suffer as few cache misses as possible, i.e. ideally both the input data read by the query executor and the outputs written by it are cached. While this is not always possible, keeping this fundamental fact in mind has helped us build a performant system.

The Read Side

Since MapD is a columnar store (i.e. data for a column is stored contiguously in memory rather than interleaved with other columns as with traditional databases), minimizing cache misses when reading data in is relatively easy. On CPU, the first time a subset of bytes of a 64-byte cache line (the native cache line width on X86) is touched, we pay an unavoidable penalty to bring that 64 bytes from CPU main memory (i.e. RAM) into cache. However, the next accesses to other sections of that cache line are relatively free. The CPU can optimize even further and pre-fetch even larger regions of memory when accesses have spatial locality, which they do when reading a full column or subset of a column from RAM. On the GPU side, it is a bit more complicated as you have many of thousands of threads working simultaneously. The best pattern for Nvidia GPUs is for each group of 32 threads (called a warp) to simultaneously access 128 contiguous bytes, so we make it such that each thread does not read memory sequentially, but rather in strides, skipping 32 data elements with each read so together the threads in a warp read a contiguous chunk of memory.

The Write Side

Writes are a bit trickier, particularly for GROUP BY (and other operations like JOINS). GROUP BY is fundamentally a "scatter" operation, meaning that we can read the inputs to the query sequentially but must write the outputs to different slots in memory according to the group generated by the input. For example, for the query "SELECT user_id, COUNT(*) AS n FROM website_users GROUP BY user_id", without sorting the input, we cannot expect user_id to be in contiguous order and thus can expect each input row to be written out to an output slot potentially distant (in memory addressing terms) from the last write. How do we optimize for this?

One approach: sorted projections

One approach, as alluded to above, is to sort the input on the GROUP BY keys before performing the GROUP BY. This is likely too expensive to do for every query at run-time, but some systems (i.e. Vertica) do generate pre-sorted projections to answer common query patterns faster. This approach also has the advantage of allowing aggressive run-length encoding of the sorted columns, although on the flip-side it can quickly lead to bloat of data stored on disk and in memory. We have not adopted this approach with MapD, since a) by leveraging the speed of GPUs we feel we can often answer queries nearly as quickly without the up-front overhead and complexity of sorting, and b) a significant portion of our use cases revolve around ad-hoc data exploration where the query patterns cannot be predicted in advance. (That said, it could be a future optimization). It should be noted that there is often pattern locality in data, particularly for columns like timestamps which are often inserted in time order as they are generated, so sometimes we get de facto sorting of columns for free.

Our approach

The other strategy for performant GROUP BY, at a very high level, is to try to minimize the footprint of the GROUP BY output slots, while also trying to ensure that more frequent groups or groups that tend to occur together get written to nearby slots in memory. Finally, we would like, to the extent possible, to minimize the compute cycles needed to compute the output slot, while simultaneously minimizing the divergence (i.e. branching) that occurs when different input groups map to the same output slot, forcing us to look for another one (hash collisions). This is a particular consideration on GPUs, since threads in a group execute in lock-step, divergence is more highly penalized.

A brief primer on hashing algorithms

To understand how we try to achieve these goals, it is useful to know a bit about hashing algorithms.

When the executor reads an input tuple, comprising the key or keys of the GROUP BY as well as the inputs into the GROUP BY aggregate clauses, it needs to decide which output slot/bin to write that group to. Of course, this routing of input group to output slot needs to be

deterministic, otherwise different tuples that belong to the same group won't be "grouped" together!

Most hashing algorithms take the space defined input keys and apply some algorithm to find a slot in a smaller output space. This is important when the potential cardinality of the GROUP BY space becomes large. For example, imagine grouping by 3 columns, each with 1M distinct values each. There are a potential 1M^3, or 1 quintillion, groups! Of course, there cannot be more groups than rows in the underlying table, and there will likely be some clustering (i.e. many tuples in a small number of groups), but the point is that up front it is impossible to allocate that much memory such that every group would have a defined slot.

MapD's "baseline" hashing algorithm

In this case, MapD, like all other databases, will likely incur many cache misses as groups are written to disparate output slots. If we make the hash "fill rate" low, and allocate more space for the output, we will have less collisions but more cache misses. Conversely, with a high "fill rate", we will have less cache misses but more collisions.

One advantage that MapD has in this regard over other databases is that it can perform scan queries very fast (a server with 8 Nvidia V100 GPUs has over 7 TB/sec (!) of bandwidth, compare that to 100-150 GB/sec on CPU). As such, when we know the cardinality of the hash space will be large, we will actually launch a pre-flight cardinality estimation scan, that uses an adaptation of the approximate count distinct algorithm (which thankfully uses very few bits and thus has relatively few cache misses) to get an idea of how many actual groups exist. We can then select the optimum fill rate for the hash algorithm, finding the right balance between cache misses and hash collisions. If you're interested, MapD uses open address hashing with linear probing, which exhibits a good balance of performance and low memory usage.

Introducing Perfect Hashing

There is only so much one can do for such cases, but the story is quite different when the group-by cardinality (i.e. number of distinct groups generated by a query) is more manageable. Rather than being the exception, we have found that at least for our target use case of exploratory visual analytics, users are often only grouping by one or two columns with relatively low cardinalities. We have optimized significantly around such cases, as described below.

When cardinalities are relatively low, it becomes feasible to create a hash bucket/slot for each possible group by slot, whether or not that slot will actually be filled by a group. This approach is called perfect hashing, since each input tuple perfectly maps to one-and-only-one output tuple. Imagine instead of the three column situation outlined above, we are only grouping by two columns, each with cardinality of 100. Such a situation is far from uncommon, as database users often want to roll up on things like US State (50 states), age (likely less than 100 distinct values), day of week (7 days), airline carrier (29 US airline carriers since 1989), or every quarter

over the last 5 years (20 quarters). In cases like this (and even with much higher cardinalities), perfect hashing can give much better performance compared to other hashing algorithms.

MapD has an extensive infrastructure in place to allow for perfect hashing of group by (and join) queries whenever feasible. That infrastructure is composed of the following three parts: range metadata retention, fragment skipping (skip lists), dictionary encoding of strings, and a sophisticated expression range calculator.

Range metadata: To perform perfect hashing, we need to know the maximum possible range of each input column so that we can allocate the correct number of slots. If we don't know the input range, we can't use perfect hashing since the algorithm assumes every possible group by slot is allocated a hash bin.

MapD stores metadata for each chunk, which is a horizontal partition (in MapD parlance, fragment) of a column. Metadata is updated on data insert/import, although in the future we could compute it lazily the first time the executor touches a column. Included in this metadata is the min and max of a column (for all numeric columns and dictionary encoded strings), which we use to bound the possible range of the column.

Fragment skipping: MapD can use the range metadata collected above to skip fragments (partitions) that can't possibly pass the predicate of the query. For example, if a user has ten years of data stored in MapD, inserted in time order, but sends a query that only filters on the last year, any fragments that have no data from the last year can be skipped. We then update the range information for the group by to exclude skipped fragments, oftentimes narrowing the number of hash output slots we need to allocate

Dictionary-encoded strings: Although MapD supports the storage and querying of raw strings, by default it dictionary encodes strings to promote fast filtering, grouping and joins. Dictionary-encoding is a technique where each unique raw string in a column is assigned a unique integer, such that "New York" might map to 1, "California" to 2, etc. It is not only much faster to hash integers than raw strings, but it has the advantage that a text column will now have a fixed range (i.e. all US states will map to integers from 1 to 50), which then can be perfectly hashed. There is also another performance advantage typically associated with dictionary encoding. Since common strings (like "California") will typically appear earlier in a dataset than uncommon strings (like "Wyoming"); this means that we will see more cache locality when hashing these strings as the common strings will hash to nearby, lower numbered buckets. A future optimization could be to allow a dictionary to be re-sorted by frequency after data import, maximizing this phenomenon.

Expression range calculation: Many SQL queries group not by a raw column but by some expression over one or more columns. For example, when constructing a histogram, the input space is mapped to a set of bins (perhaps 20). The MapD system has extensive abilities to take the input ranges of sets of columns (from the min/max of those columns, outlined above) and

compute the effect of both arithmetic and date operations on those ranges. For example, the Expression Range analyzer would be able to determine that a histogram binning operation like (a0 – aMin) / (aMax – aMin) * num_bins, will map to a range like 0 to 19. Similarly, for date operations, such as date_trunc(day, timestamp), we are able to compute the number of possible days in the space between the min and max of the timestamp column. Even though we internally store dates and timestamps as integer UNIX epochs, the expression range analyzer is smart enough to know that there can be gaps in the epoch between successive time bins (86400 seconds when binning by day for example), and reduce the calculated expression range cardinality appropriately.

The expression range calculator is also able to push down query predicates (i.e. date_column > '2017-01-01' into the expression range calculation. Hence, as a user applies increasingly restrictive filters to columns that are being grouped by, the number of slots needed for the group by decreases, allowing the system to choose a performant perfect hash (vs baseline hash) in more situations.

The upshot

With all of these optimizations, we can frequently use a perfect hash to evaluate a user's query (frequently driven by a visualization) rather than falling back to a slower baseline hash. As an example, to compute a heatmap of users by US. state by day of week would only need 50 X 7 = 350 output slots, something that would easily fit in the cache of any modern processor.

Currently MapD uses rough heuristics on when to use perfect hashing vs the default baseline (open addressing) hash. Generally, we are much more aggressive in employing the perfect hash when hashing by only a single column, as the percentage of "holes" are unused slots is likely to be lower than when grouping by multiple columns. The logic goes like this: imagine two uncorrelated columns, each with 50% of the values between their respective min/maxes filled. Grouping by two columns will then only lead to 50% X 50% = 25% of the slots being filled in the best case, and the number will likely be significantly lower due to correlation between columns (for example, between age and income).

"Cacheing" In

It is clear that in many situations perfect hashing allows us to reduce the size of the hash output space and increase the locality of writes to those bins. As a final matter, how do we ensure that our writes to the hash bins hit the cache as much as possible?

On CPU this is pretty straightforward, as an X86 processor makes this happen nearly automagically.

For a GPU, things are a bit more complex. Nvidia GPUs, although they have implicit cache, tend to perform optimally when instructed to use an explicitly managed cache called shared

memory. On current GPUs each sub-processor has access to 64KB of fast L1 cache, which means that this shraed memory can be used for group by operations on up to a few thousand groups, depending on the width of the payload (aggregate columns). We've found that when this cache can be used, close to peak memory bandwidth can be obtained, which on GPUs is quite formidable (now approaching 1 TB/sec per card).

Hopefully it is clear that aside from leveraging GPUs, there are many other optimizations we have put in place to ensure that the MapD system can query large datasets with maximum performance. As always, building a fast system often means getting into the dirty details of byte-wrangling and processor architecture, but we have found the effort has paid off in a system that we feel is faster than anything else on the market, whether running on CPU or GPU.