ACCELERATING OLAP OPERATIONS IN IN-GPU-MEMORY MOLAP DATABASES

1. MOTIVATION

Multidimensional OLAP databases allow users to analyze data from different perspectives by applying OLAP operations such as *roll up*, *drill down*, or *slice* and *dice*.

***Rolling up (aggregating)* data helps users to get a better understanding of their data by providing a summarized and compact representation. *Drilling down* into the data lets users take a look at the source data of aggregations.**

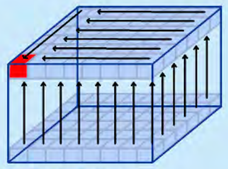


Fig. 1: Roll up

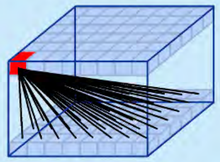


Fig. 2: Drill down

Often only a subset is sufficient to analyse big abmounts of data. *Slicing* the cube is one possibility to obtain a reduced subset of the data. If the reduction of the dataset is based on conditions, advanced dimension filters have to be applied. Turning the cube around to look at data from another point of view is called *dicing*.

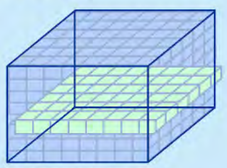


Fig. 3: Slice

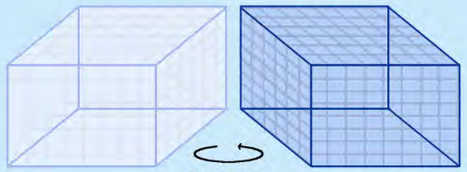


Fig. 4: Dice

Handling very large dimensions with hundreds of thousands or even millions of elements lead to datasets of billions of potential cube cells. Although usually the amount of actually stored data cells is essentially smaller, it's still quite a challenge to provide the desired information in a reasonable time.

Since most of the OLAP operations are SIMD instructions, it only seems natural to use GPUs to parallelize calculations.

In the following, a brief introduction is given on how an appropriate storage model could look like to prepare for OLAP calculations on GPU and to deal with the limited VRAM resources. Furthermore, algorithms are introduced to calculate aggregations and advanced dimension filters on GPU. Finally, the performance of these algorithms is presented in comparison to a multithreaded CPU implementation.

2. STORAGE

Storing *all data in GPU memory* has the advantage of not having to transfer it from slower CPU memory to GPU memory before each calculation, like it is done in streaming approaches. Though, the disadvantage is that the storage model has to deal with the *limited memory resources* the GPU offers.

Therefore in our approach *empty cube cells* (value = 0) aren't stored. Also *consolidated cells* aren't kept in storage but calculated on demand to save memory and to be up to date.

To *compress the path* of a cube cell, which is used to identify cells in the storage, each element of a cell path is put *bitwise with variable size* into a uint64 integer. *Bitmasks* that keep track of the maximum element sizes of each dimension are stored separately to extract the path elements out of the uint64 integer again, when the cell is later used in calculations.

If one uint64 integer is not sufficient to store the maximum cell path of the cube, one ore more uint64 integers are added. The value of a cell is stored in a double. The uint64 integers of the cell path and the double of the cell value are grouped together

Fig. 5: Only filled base cells are stored

- sorted by the cell path - in arrays with the size of one grid (#blocks * #threads per block) and then organised in a GpuPage structure. This makes it possible on the one hand that one CUDA kernel is able to calculate one GpuPage at once *without idling threads*. On the other hand, GpuPages store their minimum and maximum cell path so that a *high-level prefiltering* on GpuPages filtering out the unnecessary cube cells can take place before calculations.

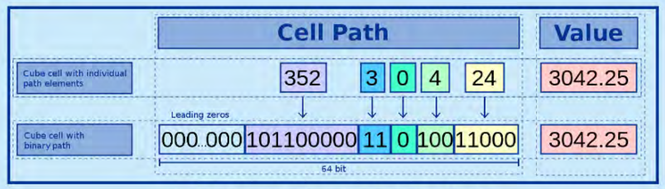


Fig. 6: The cell path is compressd by storing individual path elements in a uint64 integer

3. AGGREGATION

After the high-level prefiltering on GpuPages mentioned above took place, another pre-processing step is taken to filter out unnecessary cells on a lower level which are still left in the remaining GpuPages. Now the aggregation algorithm is able to aggregate the base cells to the according consolidated cells. The challenge hereby is - like mentioned in the first section - that the cube is usually sparse, i.e. not every possible base cell in storage exists (value = 0). Often many of the consolidated cells don't have any base cells contributing to them, which leaves them empty as well. To deal with these *large sparse areas*, the aggregation algorithm looks at all the existing base cells that the prefilter step selected out of the storage - one CUDA thread per cell. Then every dimension element is extracted out of the cell path and looked up in metadata stored in constant memory, to which aggregated elements it contributes to. Then every possible permutation of the relevant aggregated dimension elements - and thus the cell path of every aggregated cell the current base cell is contributing to - is created. The cell value of the base cell is then added with *atomic add* to the cell value of all of these aggregated cells.

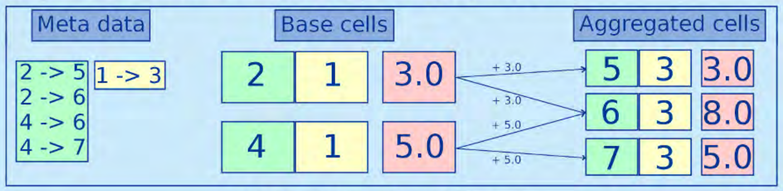


Fig. 7: Calculation of consolidated cells

The number of aggregated cells with a value != 0 is not known until the aggregation algorithm has finished. The number of potential aggregated cells is often very large, so it's usually a bad idea to allocate memory for all potential aggregated cells. Unfortunately, memory can't be allocated dynamically in kernels. This is why we use a *hash map* with an estimated size to store the result cells. If the size turns out to be too small, the progress of the current aggregation algorithm is stored and we start all over with a bigger hash map and then continue with the aggregation algorithm where we left. A nice side effect of using a hash map to store the results of the aggregation is that using more than one hash function *reduces thread contention*, which occurs when different threads write to the same aggregated cell. The disadvantage of using several hash functions is that *duplicates* of the target aggregated cells are likely to be created, which have to be removed in a post-processing step later on.

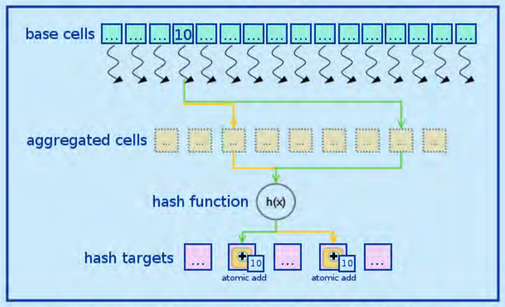


Fig. 8: Results of aggregation are stored in a hash map

4. DIMENSION FILTERS

While basic slicing is simply done by manually selecting only some of the elements in each dimension, which are then handed as parameters to the aggregation algorithm, advanced filters select elements of a dimension automatically *based on a predefined condition*, e.g.: "Select elements, where ALL cells in the according cube slice have a value greater than 1000". Or "Select elements, where ANY cell in the according cube slice has a value less than 1". These dimension filters are applied to base cells as well as consolidated cells, so at first the aggregation algorithm has to be run to create all consolidated cells in the cell area to be filtered.

All cells of this area (source cells) are then handed to the dimension filter algorithm, where for each source cell is checked, if the condition is satisfied for a certain element in the dimension to be filtered (target). If the condition is satisfied for a target element, a flag is set in a flag array with the size of the target dimension, so that other threads checking cells which contribute to the same target can immediatly return after finding out that the condition for this target is already satisfied. Some cases require to count the number of cells that satisfy the condition for a certain target (e.g. ALL > 1), so another array with the size of the target dimension has to be added to store the number of cells satisfying the condition. Like in the aggregation, the number of target elements that satisfy the condition is only known after the filter algorithm has been executed. So we use the hash map approach as described in section 3 again to avoid allocating too much memory and to reduce contention.

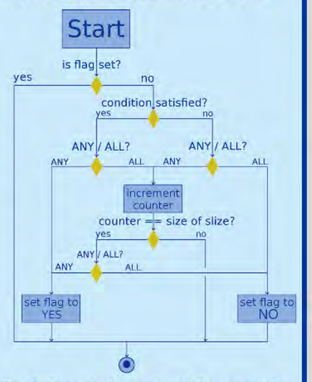
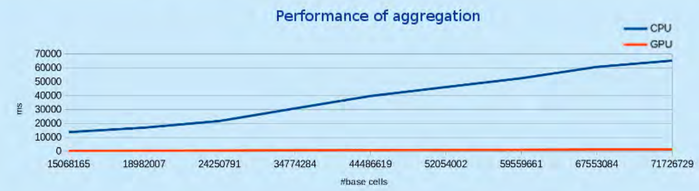
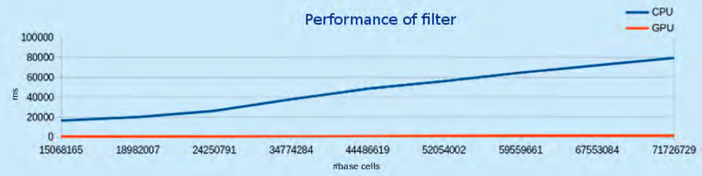


Fig. 9: target element satisfies condition if (flag == yes) || (0 satisfies condition) (empty cells are treated as 0)

5. RESULTS





6. SCAN THE CODE ...

... to download a free trial or check us out in the cloud at
www.jedox.com