PERSPECTIVE MEMORY OPTIMIZATION REPORT

Feb 2020 - Gabriel Cuvillier

Table des matières

[Current results 2](#_Toc33022569)

[Input files 2](#_Toc33022570)

[Memory profiling utilities 3](#_Toc33022571)

[Memory behavior 5](#_Toc33022572)

[Big picture 5](#_Toc33022573)

[Detailed report output 7](#_Toc33022574)

[What optimizations have been done 12](#_Toc33022575)

[1) Fixed memory leaks issues 12](#_Toc33022576)

[2) Reduced data size types of columns 12](#_Toc33022577)

[3) Do not do unneeded operations that might increase memory 12](#_Toc33022578)

[4) To not use the t\_scalar type for indexes of std::map, but use “t\_index” instead 13](#_Toc33022579)

[5) Disabled the memory resize factor of custom implemented containers during the reserve() method 13](#_Toc33022580)

[6) Release unneeded objects ASAP during algorithms, and not only at end of functions 14](#_Toc33022581)

[7) Use alternative containers implementation that are more memory-efficient (with memory pre-reservation possible) 14](#_Toc33022582)

[8) Do not maintain 2 parallel Views during a query, after a certain thresold of data size 14](#_Toc33022583)

[9) Use std::vector memory pre-reservation 15](#_Toc33022584)

[Further investigations and Recommendations 17](#_Toc33022585)

[In case of Issues 17](#_Toc33022586)

# Current results

With the latest memory optimization work, it is now possible to open the Sales9M file in Datadocs (530MB = concatenation of Sales8M+Sales1M). However, that's more a theoretical limit than a practical one: queries will not really work on such files.

So, here are the detailed and practical results, classified by query type VS max possible file size (before crash)

|  |  |  |
| --- | --- | --- |
| Query type | Max possible file size | |
|  | With optimizations | No optimizations |
|  |  |  |
| Initial load | Sales9M (530MB) | Sales4M (240MB) |
| **Context Zero query** |  |  |
| “Sort” query | Sales9M (530MB) | Sales3M (170MB) |
| **Context One query** |  |  |
| “Rows” query | Sales9M (530MB) | Sales2M (115MB) |
| “Values” query | Sales8M (472MB) | Sales2M (115MB) |
| “Rows+Values” query | Sales7M (405MB) | Sales2M (115MB) |
| **Context Two query** |  |  |
| “Columns” query | Sales5M (290MB) | Sales2M (115MB) |
| “Columns+Values” query | Sales5M (290MB) | Sales2M (115MB) |

Overall, the PSP memory requirements have almost divided by 3 between old version and the new optimized version of PSP. Initially, everything was fine under the Sales2M limit (115MB), with a max theoretical possible file loaded being Sales4M (240MB, and without doing any further queries).

Now, we can assume everything is fine under the Sales5M limit (290MB), with a max theoretical possible file size of Sales9M.

## Input files

Just a small word on input files: I've created several files based on the Sales<X>M.csv files: 1M,2M,3M,4M, and so on... up to 16M.

You just have to concatenate files using command line tools, for example to create a Sales5M file on Linux:

cat Sales1M.csv Sales4M.csv > Sales5M.csv  
  
That's it.

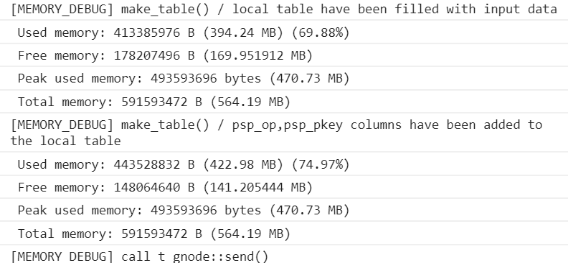
# Memory profiling utilities

There are currently no tools to debug WebAssembly memory, so the only possible way to understand memory behavior is to rely on manual "printf" code. This is a bit tedious...

To help the task, I've created a set of utilities in **memory\_debug.h :** there is now a new macro **MEM\_REPORT(x)** that will report "x" + a dump of various memory information:

* Used memory:   Memory that have been actually allocated by the program (malloc, new, stl containers, etc.)
* Free memory:    This is Total - Used
* Peak memory:    maximum memory used by the program at the time of the call
* Total memory:   Total available Wasm memory

In the Chrome Developer console, it looks like this:



This MEM\_REPORT macro is enabled/disabled with the build flag **PSP\_WASM\_MEMORY\_DEBUG**

**set PSP\_WASM\_MEMORY\_DEBUG=1**

**yarn run build**

For the purpose to help memory debugging, I've already put in the code many of these MEM\_REPORT calls, at interesting points where memory will change. So, the developer that will have the task to profile memory just build with PSP\_WASM\_MEMORY\_DEBUG, and test PSP with a file to review memory console log and spot potential weaknesses. The Dev can add many calls MEM\_REPORT, to further refine where an issue might appear = this is a bit like "memory sampling" at developer-defined time.

An example output is provided in this report, in the section 4b), with explanation of how to interpret the output.

Some preliminary remarks on the meaning of these metrics:

* Used memory means what have been reserved/allocated, but it does not mean what is really used by an algorithm. For example, when a std::vector is resized automatically, it will reserve space that might not be used if the vector is not fully filled in the end. But this case does not happen very much in practice now (I've changed a couple of things on this side), and Used Memory should closely match what is really used.
* Having a high amount of Free memory does not mean we can do all kind of allocations possible. If memory is highly fragmented (due to many small allocations/deallocations done previously), maybe there is not enough "contiguous" space to allocate for example a 30MB buffer. Memory fragmentation is an issue in PSP, due to many usage of std::map type containers: they do many allocations of small objects.
* Also, a low level of Free memory is not dramatic either, providing Total memory is below 2GB. Upon an allocation, the Total available memory will grow to match the need (hence increasing free space).
* Peak memory is an **extremely** interesting metric: it allows to spot where big allocations might have occurred during of an algorithm, between two calls of MEM\_REPORT. A value approaching 2GB shows that the program almost reached the possible limit. By keeping a close eye to this value, it is possible to iterate (= reduce the interval between the MEM\_REPORT calls) and then spot precisely where in the code the memory is being really allocated. I've found all the weak spots like this!
* Also, a peak memory is not 100% accurate (+/- 5MB I think), but it should be close to reality.
* Finally, Total Memory will grow as needed automatically, the maximum possible value for WebAssembly being 2GB. Once at this value, this mean that we triggered the memory limit. If the free memory at this point is low, things will soon crash :)

# Memory behavior

## Big picture

PSP is really memory intensive, doing a lot of copies of the data for various reasons.   
Here are some general rules of the memory behavior, using the optimized version:

1: Let's assume the **input CSV file size is** "**X**"

2: Ingest code:

        The Arrow buffer in memory will have a size of **X** on average.

3: PSP "Load Arrow Buffer" to Table:

        Based on the Arrow buffer, the initial Table load is split in several parts:

        - A first part where PSP just load the columns in memory

=> Memory will grow to X

        - A part where various columns are being added for internal reasons (psp\_key, psp\_op)

=> Memory will grow to X+15%

        - A part with a copy of the table to internal PSP data structure (gnode)

=> Memory will grow to X+20%, with peak to 2\*X

        - A successive list of various operations: flatten, new copy of the data to another internal data structure (gnode\_state), and creation of a "mapping"

=> At this point, used memory will grow to **X+30%**, with a peak to **3,3\*X** (mostly due to various copies along the way)

While it might sound a lot, this is in fact the part I optimized the most. With the un-optimized version, the used memory was 4\*X (vs \*1.3 now), and peak was 6\*X (vs \*3,3 now).

4:    PSP "View creation"

        Then, come the creation of the first "View". Here come the queries.  
  
        There is actually 3 kind of queries:  
        - Context Zero (no particular attributes, for example just “sorting” a column)  
        - Context One  (Row based, and/or Value based)  
        - Context Two  (Column based)

Right after a Table is created, this is a "Context Zero" that is being created.

In the end, memory grows to **X\*2.75**, with a memory spike to **X\*3,7**

And now, you have your first Grid displayed to the user.  
  
We can then understand why the Sales9M file is the max limit as of now: as the max available memory for WebAssembly is 2GB, that basically means that the max possible file size is something like 2GB/3,7 (due to peak memory usage ratio) = 550MB ~= the Sales9M file.  
   
In this situation, only the "Context Zero" query type have been taken into consideration (initial load + first view, or sorting a column). Other types of queries have much more memory requirements, explaining things start to go unstable.

Here is a simplified overview of what can be expected in terms ratio between input CSV file size and Peak used memory size, per Query type:

|  |  |
| --- | --- |
|  | Peak Memory ratio (= to apply to CSV file size) |
| Context 0 query: first view after initial load, or sort a column | 3,5x |
| Context 1 query: Group by Row <Attribute> | 4x |
| Context 1 query: Count of <Attribute> | 4,5x |
| Context 1 query: Group by Row + Count Of | 5x |
| Context 2 query: Group by Column <Attribute> | 6x |
| Context 2 query: Group by Colum + Count Of | 6,5x |
| Context 2 query: Group by Row + Group By Column + Count Of | 7,5x |

Without any kind of optimizations enabled, these ratios are multiplied by 3.

It should be noted that in memory you will have at least :

* + - 1: The Table Data
    - 2: The result View Data

Put it in other words, there is at least twice the input CSV file Data.

During a View computation, there is the following additional data:

* + - 3: The NEW result View Data (in addition to the previous one, that will be released once the new view is OK)
    - 4: Internal Data used by the View computation itself (which generally have the same size as the result View data)

It should be noted that an optimization have been done to mitigate against the accumulation of items 2 and 3: now, above a certain threshold of data size, when a new result View is being computed, the previous View is immediately deleted before entering computation.

In the end, the theoretical minimum peak memory the program can work with is at least 3 times the input data size : that is, items 1+2+4. As WebAssembly is limited to 2GB, this theoretically limits input data size to something like 670MB = the Sales11M file.

## Detailed report output

Here is a sample memory report output of the "Sales6M.csv" file, as how it can be observed if ENABLE\_MEMORY\_DEBUG is activated (and memory optimizations on), with some explanations.

In *italics,* the console output.

**Initial file: Sales6M.csv - 348MB**

*ArrayBuffer(355538722) {}*  
  
  => this is the size of the Arrow buffer = 339MB. This is roughly the same size as the input file size.   
      
*Arrow buffer exceed 256MB  
Entering memory pressure mode*  
  => this mean the Arrow buffer size is big, and then we will now delete the views as soon a query is done (see last part of section 5 about this optimization)

*[MEMORY\_DEBUG] call make\_table()  
Used memory: 5362392 B (5.11 MB) (31.96%)  
Free memory: 11414824 B (10.88 MB)  
Peak used memory: 5284960 bytes (5.04 MB)  
Total memory: 16777216 B (16.00 MB)*  
  
      => this is the initial memory layout of PSP, before anything else have been done: 5MB of memory used (the C Stack in fact), with 16MB of total available memory

*[MEMORY\_DEBUG] make\_table() / local table have been created and extended to the number of rows  
Used memory: 353373728 B (337.00 MB) (91.84%)  
Free memory: 31388128 B (29.93 MB)  
Peak used memory: 353293296 B (336.93 MB)  
Total memory: 384761856 B (366.94 MB)*

      => this is when a Table is created in memory to match the upcoming Arrow buffer loaded. We can observe a +334MB memory increase, roughly the size of the Arrow buffer. That's good!

      => Note that the Total memory have increased as well as peak memory.

*[MEMORY\_DEBUG] make\_table() / local table have been filled with data  
Used memory: 353385560 B (337.01 MB) (76.53%)  
Free memory: 108381096 B (103.36 MB)  
Peak used memory: 433595376 B (413.51 MB)  
Total memory: 461766656 B (440.38 MB)*  
  
  => Filling the table with data from Arrow buffer did not caused used memory increase. Notice however the Total memory have increased, but there is more free space too, and peak moved a bit, but not that much. It probably means that temporary allocations were done, but properly cleaned up. Good too.

*[MEMORY\_DEBUG] make\_table() / psp\_op,psp\_pkey columns have been added to the local table  
 Used memory: 383518712 B (365.75 MB) (83.05%)  
 Free memory: 78247944 B (74.62 MB)  
 Peak used memory: 433595376 B (413.51 MB)  
 Total memory: 461766656 B (440.38 MB)*

      => this is the add of two columns used by PSP engine internally (unique key and operation, for each row) = + 28MB of data. Ok too, with such a huge data set, adding two new columns, this is to be expected

*[MEMORY\_DEBUG] quit t\_gnode::send() / table have been cloned  
 Used memory: 838927816 B (800.06 MB) (96.11%)  
 Free memory: 33946168 B (32.37 MB)  
 Peak used memory: 848892912 B (809.57 MB)  
 Total memory: 872873984 B (832.44 MB)*

      => For some reason, the created table is only a temporary one, and have to be cloned in a PSP "gnode".

    => Notice the memory behavior: used memory is now a bit more than twice than in the previous step: this is due to having old table + cloned table.

*[MEMORY\_DEBUG] make\_table() / local table have been released  
 Used memory: 460903184 B (439.55 MB) (52.80%)  
 Free memory: 411970800 B (392.89 MB)  
 Peak used memory: 848892912 B (809.57 MB)  
 Total memory: 872873984 B (832.44 MB)*

      => Hopefully, the temporary table is released soon after (NB: it was not the case immediately without the optimizations I did).

      => So used memory come back to 439MB. Note the memory use difference between the new table (439MB), and the old table (365MB). For some reason there's a +74MB overhead. I don't understand why exactly, as things are supposed to be exactly the same. Maybe some internal PSP metadata stuff added ?

*[MEMORY\_DEBUG] t\_gnode::\_process() / the table is going to be flattened*  
      => For some reason, the table is going to be flattened. I don't get why, but here it is: a new copy will soon be done

       => Question: does the flatten is really needed at this point ? I don't think so, but anyway...

*[MEMORY\_DEBUG] t\_gnode::\_process() / flatten completed  
 Used memory: 874925752 B (834.39 MB) (69.18%)  
 Free memory: 389787976 B (371.73 MB)  
 Peak used memory: 1163928560 B (1110.01 MB)  
 Total memory: 1264713728 B (1206.12 MB)*      => The flatten is completed, and as so, there is again the double of data in memory: unflattenned data AND flattened data. This explains the 834MB we are now (= ~2\*the initial table data size + the 74MB metadata overhead)  
    
      => Note the Peak memory usage... 1.1G. The flatten did certainly many temporary allocations.

*[MEMORY\_DEBUG] t\_gnode::\_process() / initial table have been released  
 Used memory: 419516696 B (400.08 MB) (33.17%)*  
  => Hopefully again, the unflattened table is being released now. So, memory come back to normal levels: 400MB, which is more or less the initial table size before flattening. The metadata overhead have been released along with the unflattened table. That's good.

*[MEMORY\_DEBUG] gnode\_state::notify() / table is going to be cloned again*  
  => The table will be cloned again !  
    
  => This time, the table will go from "gnode" to "gnode\_state" data structure. I am not sure why, but that's how this works.  
    
     *[MEMORY\_DEBUG] gnode\_state::notify() / table cloned into gnode\_state  
 Used memory: 833527528 B (794.91 MB) (65.91%)*  
  
  => And the result: we have both the flattened table in memory, and its clone in gnode\_state = 794MB = ~2 times the table size.   
    
Note the complexity of the algorithm just to load a table in the end :

Arrow buffer => load => temporary table in memory => copy-then-release => table in gnode => flatten-then-release => temporary table in memory => copy => table in gnode\_state  
      
  => But there, instead of immediately release the temporary flattened table, things gets a bit worse before the actual release of temporary flattened data.

*[MEMORY\_DEBUG] gnode\_state::notify() / m\_mappings reserved  
 Used memory: 881526008 B (840.69 MB) (69.70%)*  
  => For some reason, a "mapping" of "psp\_key" column to "row indexes" is getting created and stored persistently in memory. This used to be very memory heavy, but with latest optimizations the impact is moderate (+45MB vs +400MB initially!).

*[MEMORY\_DEBUG] make\_table() / flatenned table now released  
 Used memory: 467503280 B (445.85 MB) (36.97%)*

      => Hopefully again, the temporary flattened table is being released. We now just have the final table in the gnode\_state = 445MB

*[MEMORY\_DEBUG] quit make\_table()*  
  => Ok, now the Table is fully loaded in memory  
  
  
    And now, starts the first View creation, and a first Query: a "Context Zero" query  
  
*[MEMORY\_DEBUG] call make\_view\_zero()  
[MEMORY\_DEBUG] call make\_context\_zero()  
[MEMORY\_DEBUG] call t\_gnode::\_register\_context()  
[MEMORY\_DEBUG] call t\_ctx0::notify()*    
      => nothing special seems to happen there. Memory stays the same.

       => note the 't\_ctx0' => it means we are in a "context zero" query

*[MEMORY\_DEBUG] quit t\_ctx0::notify()  
 Used memory: 587494216 B (560.28 MB) (46.45%)*

      => It looks like many things have happened again, possibly the table data have been partly cloned in the View/context, in a structure called **flat\_traversal**. It looks like a View will always have a partial duplication of the table data, for the purpose of "traversing" it.

*[MEMORY\_DEBUG] ftrav::step\_end() / after allocating new indices  
 Used memory: 923494368 B (880.71 MB) (65.98%)*  
  => Now 880MB of used memory, after allocation of some temporary data structure

*[MEMORY\_DEBUG] ftrav::step\_end() / some memory just have been cleared  
 Used memory: 803494256 B (766.27 MB) (57.41%)*  
  
  => Things are hopefully now cleaned up a bit: 115MB released

    => Also, initially the algorithm did some unneeded additional things that I disabled (computation of an unused minmax value for each column, doing a clone of each columns...).

*[MEMORY\_DEBUG] quit ftrav::step\_end()  
 Used memory: 973834360 B (928.72 MB) (69.58%)*

  => But nevertheless, new things are being computed in the flat\_traversal data structure: 928MB of used memory in the end.  
  That is: 2 times the size of the PSP table in memory. It looks like the View will in the end have the full data cloned inside.

*[MEMORY\_DEBUG] quit make\_view\_zero()  
 Used memory: 973835016 B (928.72 MB) (69.58%)  
 Free memory: 425817336 B (406.09 MB)  
 Peak used memory: 1298891760 B (1238.72 MB)  
 Total memory: 1399652352 B (1334.81 MB)*

  => And finally, the Grid is now displayed to the user.  
    
  => We have 928MB for both the loaded Table + the View, with a peak of 1.24GB memory used.  
    
  Hence, used memory is 2,7x the size of input CSV file, and peak is almost 3,7x.

**This practical memory peak observation actually limits the max file size to 2GB/3,7 = 550MB => the Sales9M file.**

Now, let's do a simple "Row+Value" query (Group on “Territory ID” + count of "Row"). NB: this is a "Context 1" query

*deleting the view due to memory pressure*  
  
  => The current view is deleted to reclaim some memory  
    
*[MEMORY\_DEBUG] call make\_view\_one()*

*Used memory: 467478008 B (445.82 MB) (28.27%)*

  => We can see there is no memory leaks => we have the 445MB of PSP table in memory (+ the "mappings"),and 483MB of View data have been properly cleaned up.  
  Initially, there was a memory leak there, and the view was not released...  
  
*[MEMORY\_DEBUG] quit t\_gnode::\_register\_context()  
Used memory: 852286160 B (812.80 MB) (45.15%)  
Free memory: 1035216176 B (987.26 MB)  
Peak used memory: 1786721264 B (1703.95 MB)  
Total memory: 1887502336 B (1800.06 MB)*  
  
  => The view is finally computed, and data show to user.

While it worked, and we are now at 812GB memory used, you can see **that the Peak Memory usage have been reaching 1.7GB.**  Just 300MB below the max authorized limit.

# What optimizations have been done

I've implemented many things to reduce memory overhead. Most of them are indicated with a comment in the code, starting with // GAB: memory optim  
  
Here is a list, grouped by "strategy":

### 1) Fixed memory leaks issues

* Fixed memory leak with ASYNCIFY and make\_view\_xxx functions: a function using resolve() to return a smart pointer should NOT return the smart pointer through the return statement (it will never be released). This was making Views to leak and never be deleted between two queries

          File is emscripten.cpp, functions make\_view\_zero, make\_view\_one, make\_view\_two

* Fixed memory leak in JS side of Perspective, with "slices" accidentally never released. A slice keep a pointer to some data in PSP, preventing the data to be released between two queries

          File is perspective.js, function "to\_format", line 649

Both of these fixes were critical for the rest, as otherwise, each time a query was done a lot of memory was wasted (after 3 or 4 queries one the Sales1M file, things were crashing...).

### 2) Reduced data size types of columns

* Reduced the “Rownum" columns data size types in Ingest, with new column type Integer32 (initially it was Integer64).
* Removed the automatically generated column "psp\_okey" in PSP => it is used nowhere

          File emscripten.cpp, line 3370

* Removed the automatically generated column “count” in Ingest, => replaced by usage of “row” column
* Disabled "nullmap" of automatically generated column "psp\_pkey"

          File emscripten.cpp, line 3369

* Implemented Integer column data type “shrinking” in Ingest: using Integer32, Integer16, or Integer8 data types when possible instead of always using 64bits Integers.

### 3) Do not do unneeded operations that might increase memory

* Do not compute minmax of a column after having computing a View with no attribute (initial load of a Tale) => the result value is not used anywhere in the code. NB: this not related at all to "min" or "max" computation on "Values" queries.

          File is context\_zero.cpp, function t\_ctx0::step\_end, line 63  
        
          Memory usage during this minmax computation is incredibly high, due to clone of each column data for unknown reason. This mostly prevented to just load Sales6M and Sales7M files.

* For the "Context Zero" queries, there is now an optimization to prevent storing a big temporary map of precomputed values... that will be used only once afterward.   Now, the values are not precomputed and stored, but computed "as needed" (so no need to store ALL the precomputations. As they are computed only once, there is no performance impact).

          File are flat\_traversal.h line 33, and flat\_traversal.cpp, in function t\_ftrav::step\_end, line 259   and function t\_ftrav::add\_row, line 329  
                      
          This one is a bit difficult to explain, but note in the code the replacement of a "map" container by a "set" container, that is, with no associated value. The associated value, instead of being stored in the map (which is quite big!), will be computed as needed later on in step\_end method. Memory gain is interesting there, since the map may be huge.

### 4) To not use the t\_scalar type for indexes of std::map, but use “t\_index” instead

The t\_scalar type is a really large data type (32 **bytes**). Frequently in the code, some std::map or std::set containers are using this type as the key, just for the purpose of storing “PSP primary keys” of rows (the “psp\_pkey” column). PSP keys are just integers (4 **bytes**), for which the typedef “t\_index” might be used (an int32), which is quite smaller than t\_scalar (8 times smaller!)

This is one of the most memory efficient optimization technique found. The only drawback is that it prevents having a user-defined column to be used as PSP keys (because it might be of another type than Integer, such as String). Only PSP-generated keys can be used now. However, having a user-defined PSP key column is a feature not used by Datadocs application, so that’s fine.

This type replacement been done in several places in the code:

File are flat\_traversal.h line 33 and 34 (the “t\_pkeyidx\_map” and “t\_pkselem\_map” types, indexed by PSP keys), and flat\_traversal.cpp line 49

File are gnode\_state.h line 34 and 35 (the “t\_mapping” type, indexed by PSP keys), with many places in the .cpp where there is the need to convert a t\_scalar to a t\_index, for the purpose to access the container contents. The code is usually something like (in **bold**, the added part)

t\_scalar pkey = …

container[pkey**.get<t\_index>()**]

### 5) Disabled the memory resize factor of custom implemented containers during the reserve() method

* In the "t\_lstore" class, which is the backend storage class for columns: initially, each time a reserve() was done, an overhead of +20% reserved memory was added (this is to handle the case of later append to the column, but this almost never happens). Now the resize factor is only applied during push\_back if a resize is needed.

        File is storage.cpp, in function t\_lstore::reserve\_impl, line 343 (the resize factor is not applied) and function t\_lstore::push\_back, line 477 (the resize\_factor is applied there instead).  
          
        Basically, **this reduced the overall memory consumption by 20%** !

### 6) Release unneeded objects ASAP during algorithms, and not only at end of functions

This action is done in many places, and in particular in functions with big allocations or "table clones": as soon as a vector/map/whatever object is no more needed by the code, I release it immediately (not delaying the destruction at the end of function). This allow to free up some memory DURING functions, helping a lot when we arrive near the limits of memory.  
      
    Example places where this have been done:  
          
      emscripten.cpp/make\_table()   
        release the local temporary table ASAP (using enclosing blocks surrounding its usage), that is, before calling gnode::process which is memory intensive (initially it was done after). This helped to load all the files above Sales5M.  
          
      flat\_traversal.cpp  
        Almost everywhere in the step\_end() method! (there is a line of comment each time), which does many allocation of temporary-but-used-once containers.  
          
      gnode.cpp, in function t\_gnode::\_process, line 403  
         Notice the call to release\_inputs() => this will release the flattened table ASAP, before calling update\_history(), which is memory hungry (with the "mappings" stuff)

### 7) Use alternative containers implementation that are more memory-efficient (with memory pre-reservation possible)

This is done mostly in the file gnode\_state.h, by replacing **std::unordered\_map** with **boost::flat\_map** for the “mappings” attribute. This is done because the number of items that will be added in the map/set can be known in advance AND the inserted values are ordered.   
    
Rationale: std::map containers are memory hungry, and exposes to memory fragmentation issues (many small objects allocated everywhere), while the boost flat\_map is backed by a contiguous vector that takes only one big chunk of linear memory. But caution there: if the flat\_map isnot reserved in advance, it may consume in the end more memory due to the big chunk of memory being resized. Also, if the inserted values are not ordered, the performance drop would be dramatic. Hopefully, the values inserted in the mappings (the PSP “Keys”) looks like to be ordered.

### 8) Do not maintain 2 parallel Views during a query, after a certain thresold of data size

When a query is done, we keep by default the previous View in case the user decides to cancel the new View creation (so we can restore it). Only if the new View is finally computed the old one will be released. While this sound smart, this is extremely memory consuming, as views takes a lot of memory space when data set is big.  
  
Now, above a certain threshold of input data size (empirical 256MB size, that is a ~4M file), the current View is deleted as soon as a query is done.  
    
  Advantage:  almost 1/3 available more memory! Making possible to process all files above or equal Sales4M...

  Drawback:   no more possible to cancel...

  Drawback mitigation: this optimization is only enabled starting at the 4M file (224MB of Data). As such files were previously not possible to process at all, this is better than nothing ;)

This optimization proven to be the very memory efficient !

File is action\_element.js line 327 (note the "memory\_pressure" attribute), and perspective\_viewer.js line 687 ( here we delete the View as soon as possible if we are in "memory pressure").

### 9) Use std::vector memory pre-reservation

This is a very interesting memory optimization: the idea is to try to precompute the size of containers before doing a push\_back to them, in order to prevent hidden resizes that might be **extremely** costly for memory usage (the vector is resized by a factor 2 each time it is full!).

This is done there for example: t\_gstate::read\_column

But also in many other places in context\_zero, context\_one, context\_two, sparse\_tree, flat\_traversal files. Just look for the // GAB: memory optim comments

The general idea is to do the following:

If you see the following pattern:

std::vector<…> myVec;

For (condition)

myVec.push\_back(stuff)

Try to change it to:

std::vector<…> myVec;

t\_index count = 0;

For (condition)

count++

myVec.reserve(count);

For (condition)

myVec.push\_back(stuff)

This can be very memory efficient if you can know in advance the size of the vector, as it will prevent unexpected vector resizes that are wastefull for memory, in particular when vector starts to be large (in extreme corner cases, a 300MB vector might be resized to 600MB just for the purpose to add a couple of additional items…).

The minor drawback is that you will have to do a “first pass” to count the number of elements. But this should not alter the algorithm complexity (so performance impact will be negligible), and the memory gains can be very high. As so, it is a good tradeoff to do.

# Further investigations and Recommendations

As you may have noticed, the optimizations have been done mostly on the Initial Table load and “Context Zero” query types, but there is definitively room for improvements in “Context One” and in particular “Context Two” query types, which seems to have very high memory spikes (up to 7x time the input data size).

This can be explained by the usage of a complex data structure called a “sparse\_tree”, that is probably not quite memory optimized. Using the various techniques mentioned in previous section, it should be possible to decrease the memory usage by an order of magnitude.

In particular, the techniques 4) and 9) (respectively: “replace t\_scalar used as indexes of std:: containers by t\_index” and “std::vector memory pre-reservation”) are the ones to focus on in my opinion. They proven to be very efficient at this point.

An hint to look where you can apply the 9): just look at all the push\_back calls in the code => does the target container have been properly “reserved” first ? I’ve done a first pass on them (including in “sparse\_tree” data structure, and context\_zero/one/two files), but there are a lot of remaining cases.

The technique 7) (“alternative containers implementation”) might also prove to be interesting (as it was for the “mappings” of PSP indexes to row indexes, which divided by 10 the memory requirements for the container), but caution: this is only to be done when key insertion will be done ALWAYS in natural order !

You may look for std::unordered\_map and std::unordered\_set in the code, and see if you can do something interesting

NB: there is some place where it is not suitable, for example in flat\_traversal.h : do not try to replace the t\_pkeyidx\_map with a boost::flat\_map. You’ll come in big performance issues (I know, I tried!) when columns are sorted in reverse order (= keys will be inserted in reverse order in the boost::flat\_map container => this is the worst possible case, bringing a N^2 complexity for insertion).

The technique 6) is also interesting: during memory intensive operations, try to cleanup/release things as soon as they are not needed. I’m sure it could be applied somewhere in Context\_One/Context\_Two and Sparse\_Tree files/functions. I’ve did my part in Context\_Zero/Flat\_Traversal files already.

## In case of Issues

In case of issue or weird behavior with the optimizations, a particular one might be disabled as it is the most “controversial” one: this is the 7) (alternative containers implementation). Just fallback to std::unordered\_map and std::unordered\_set, and see if things behaves better.