PostgreSQL. Mission: Retrofit Memory Manager

Agents Bènoît Hüdsön and Dávid MákWhértër
Carnegie Mellon University School

Abstract
Management of heap memory in modern DBMS is something that often falls by the wayside. While academics focus on replacement policies for bufferpools, both commercial and noncommercial systems largely largely ignore the penalties for using too much heap memory during query processing. The result, particularly under heavy loads, are systems that thrash virtual memory, force evictions from operating system caches, and otherwise exhibit suboptimal performance. Our research involves wrangling in the monster of heap allocation, using a simple control strategy, to reduce or eliminate these problems and improve overall system performance and throughput.

1 Motivation
Databases are fickle creatures. As everybody well knows, DBMS performance depends on thousands of parameters, ranging from bufferpool sizes, to disk speed, to checkpointing frequency, to Ralph Nader’s current presidential campaign strategy.

One of the challenges of DBMS design is handling memory management. Operators can all use different, variable, and hard-to-predict amounts of memory. The situation is complicated by the fact that there are a variable number of clients in the DBMS, and each is issuing unpredictable sets of queries. How to optimally manage the memory available to transactions and query operators is quite difficult.

PostgreSQL, as well as other DBMS, deal with the problem of memory management by putting it to the operating system. Each client simply malloc’s and free’s the memory that it needs to execute. Virtual memory is employed to ensure that the system still functions even when too much memory has been allocated. Unfortunately, the virtual memory subsystem is completely unaware of the DBMS internal state, and cannot make intelligent decisions as to which memory pages are not immediately needed. Thus, the system is likely to experience undesirable performance due to virtual memory swapping.

Making the situation even worse is the problem that in many modern applications, such as dynamic web pages based on database-generated content, the system load can be extremely variable. It is well-known that Internet traffic exhibits hot-spots, where sites experience unbelievably high transient system loads (eg: “the slashdot effect”). During these periods of high-load, systems such as PostgreSQL can exhibit extremely high memory pressure and virtual swapping. While some connections could simply be dropped via admission control, that is clearly detrimental to clients. We would like to see the system at least attempt to make forward progress, based on what the true current memory demands of the system are.

While some DBMS may make very detailed and very controlled heap memory allocation decisions, and may be able to respond better to excessive memory pressure, there are still problems for those who currently have careless memory allocation decisions. Since memory allocations are dispersed throughout the entire system source code, the software engineering cost of carefully allocating and managing memory is prohibitive. What is needed is a simple mechanism that can be retrofitted onto these systems with uncontrolled memory management, to enable them to perform well even in the context of heavy memory pressure.

Our thesis is that in a system such as PostgreSQL, individual process’s memory allocations can easily be authorized and governed by a centralized memory management system. This management system can easily maintain approximate knowledge about each process’s current memory usage. Based on this knowledge, the management system can be used to control memory pressure by authorizing or rejecting individual process’s memory requests. Thus, with a local modification to the DBMS source code, the memory system may approximate the performance of a system whose memory management has been carefully designed and accounted.

2 Prior Work
Memory management is a common problem addressed in DBMS, and nearly all software design. Unfortunately, the knowledge and solutions discovered rarely make it out into
the database research literature.

Conventionally, academics consider only a managed bufferpool of pages that reflect data typically stored on disk, and occasionally temporary relation pages. Almost all existing DBMS research is limited to understanding the effects of page replacement and page allocation within the bufferpool. Understanding the heap-memory needs of DBMS processes, for handling relation sorting, hashing, and other operations, is far less popular.

Commercially, nearly all relevant DBMS currently make use of very simplistic heap memory management techniques. PostgreSQL, as indicated above, makes heap allocations to the operating system without reservation. Some limits are placed on each process on the sizes of sort memory buffers and such, but in reality, individual processes’ memory usage can grow far beyond those limits. MySQL and the Shore storage manager both follow the same model as PostgreSQL. The problem with these solutions is that the processes can easily grab more memory than the system can provide.

IBM DB2 takes a slightly different approach than PostgreSQL, and divides heap memory statically for different applications. Within each application, each connection and process may freely allocate memory within that heap. Thus, aside from per-application breakdown, it is very similar to the PostgreSQL implementation. This approach also suffers from the problem that if 99 applications are idle and using no memory, and one other application needs more than its configured allotment, it cannot proceed.

In summary, our work is one of the first research contributions to focus exclusively on DBMS process heap allocation management, as opposed to bufferpool management. Furthermore, we focus on solutions which dynamically manage heap memory. The consequence ensures that (a) memory is not being wasted when it could be put to better use helping a process, and (b) the system does not use too much memory and introduce swapping and other performance hits.

3 Approach

While the implementation of our system is not the primary goal of this research, it is valuable to learn from the architecture of our memory management system. We believe that our implementation decisions are able to be adapted and employed when modifying other existing systems.

The memory manager maintains an allowance for each process. As processes use memory, they make sure to maintain their actual usage at all times to be below their allowance. If they want to allocate more heap memory, they must apply for an increase in allowance. When they release memory, they can notify the memory management subsystem and reduce their allowance. When a process applies for more allowance, the memory manager can decide to grant the request, block the requester until more memory is available, or abort the transaction. In this way, we can control how much memory is used by the database processes and keep it at a level that minimizes virtual memory thrashing and interfering with OS-side buffering.

3.1 Parameters

There are four tunable parameters in our system:

- **MAX_ALLOWANCE**(integer): The maximum allowed total memory usage across all processes.
- **INITIAL_ALLOWANCE**(integer): The amount of memory a process must request when starting.
- **GROWTH_INCREMENT**(integer): The minimum amount by which a process will request to change its allowance.
- **policy**(abort or block): What to do when the request cannot be fulfilled without exceeding **MAX_ALLOWANCE**.

We have not determined the optimal settings for the parameters, although we have some intuition.

It seems intuitive that the maximum allowance should be about the amount of free RAM in the system after the other processes have been accounted for (kernel, web server, etc). There is some argument for setting it higher than available RAM: the allowances are an upper bound on the memory actually in use. However, there is an equally good argument for setting it slightly lower: any space not used by processes can be used for OS-side file buffering and caching. An additional consideration, independent of performance issues, is for systems that run a database as a background process. Such a system may want to reserve most of its memory for the important things it runs (the user’s web browser and email program, for instance) and disallow the database from using all available memory.

At the start of a transaction, a process applies for a chunk of memory of size **INITIAL_ALLOWANCE**. This allows for some admission control: if it’s unlikely that the process will be able to allocate enough memory to run, we may as well block it from running until more memory frees up. One good setting of **INITIAL_ALLOWANCE** would be analogous to the hot set buffer management policy [2]: we determine exactly the peak memory usage of the process and give it that much. This means that the process will block until it is guaranteed to be able to run to completion (or at least not be blocked for lack of memory). In our system, we are not so clever, and instead use a fixed **INITIAL_ALLOWANCE** for all processes. If all processes in the workload have similar memory requirements, then it would make sense to use the median or mean peak memory usage as the initial allowance. In a workload like TPC-C, however, different classes of queries have differing (factor of over 4) memory requirements. In this case it’s less clear what an good setting is.

Similar considerations hold for the **GROWTH_INCREMENT**. If a process has exceeded its initial allowance, the **GROWTH_INCREMENT** would ideally bump up the allowance to the peak memory usage.
In a system with classes of queries, the growth increment would ideally be the difference between the mean or median peak memory of each class. However, there is an additional consideration for the **GROWTH_INCREMENT**: if enough memory is freed, then we want to release some of our allowance. In order to avoid thrashing the allowance, a process only relinquishes one **GROWTH_INCREMENT** when it has enough unused allowance to still have a free **GROWTH_INCREMENT** left—that is, a process only shrinks when it frees two **GROWTH_INCREMENT** of memory.

The policy considerations are discussed later in the experiments section 5, as are the precise settings we used in the experimental validation.

### 3.2 PostgreSQL implementation

In PostgreSQL, all heap memory allocation is done using `palloc` and a `MemoryContext`. There is a corresponding `pfree` but usually deallocation is done *en masse* by deleting an entire memory context associated with an operator. The `MemoryContext` in turn uses the standard `malloc` and `free` to allocate and free large chunks of memory at a time. Any system that uses the same function for all heap memory allocations and frees is easy to retrofit in a similar fashion to here.

We modified the `MemoryContext` code so that just before every `malloc` and just after a sequence of `free`, the process updates its current total memory usage. If this exceeds the process’ allowance, or is sufficiently below the allowance, it must now use the memory mangement subsystem. We want to avoid accessing the subsystem if no allowance change is needed, since it is shared among processes and thus requires locking.

The count of total memory usage is slightly off. Each process carries with it some additional baggage – in particular initialized global data (that is, pages of the `.data` and `.bss` sections that have been changed in memory).

In addition, while we’ve conflated the two elsewhere in the paper, in PostgreSQL, the concept of a process and that of a transaction are different. One process may run multiple transactions, and a process may be in a state where it is not currently processing a transaction. This saves the database server from spawning off a new process for every transaction, in the case of clients doing multiple transactions. In our implementation, when a process is not in a transaction, it is allowed to freely allocate memory; doing otherwise was causing difficulties.

Ideally, before even spawning the process, the database server would check against the memory manager whether there was room for the new process. If not, it would delay the start of the process. However, this would have been difficult to integrate into the current PostgreSQL process startup procedures, and the difference is fairly minor for the number of processes we are considering.

**Interfaces:**

Internally, we used the following interface to implement the two policies (abort and block). This interface allows for a wide range of policies that address issues we did not attack, and probably for issues we did not even think of. Unfortunately, the interface is overkill for the abort policy: it should never need to unblock someone since it always aborts instead, so the last two functions are superfluous.

- `bool should_block( int request )`
  The process is asking to increase its allowance by request. Should it block or not? We also check for aborting here; if we decide to abort, the call will not return.

- `xact who_to_unblock( )`
  Check whether there is sufficient global allowance left to wake someone up that is waiting on memory (if anyone is). If so, decide who should be unblocked.

- `xact who_to_force(xact *who, bool *abort)`
  There is a deadlock that involves a memory-blocked process (see 4.1). Force a resolution. The function returns a transaction to wake up, and a flag of whether that transaction should be told to progress (and thus exceed the maximum allowance), or abort. This function does not apply to the abort policy as we wrote it, but conceivably we could block processes on memory and only abort them if, as here, push comes to shove.

### 4 Implementation Difficulties

Implementation of a memory allocator that blocks processes to wait for available memory turned out to be the most difficult part of the project. The problem is due to the fact that transactions that block on memory requests may in turn block other transactions in the system.

In PostgreSQL, transactions blocked on memory (“mem-blocked transactions”) can block other transactions in two ways. The first, is that the mem-blocking transaction is holding a heavyweight database lock needed by an incompatible transaction. The second is that the mem-blocking transaction may be holding a lightweight lock (eg: a spin-lock).

#### 4.1 Heavyweight locks and deadlocks

The heavyweight lock problem above first appears to be solved by the deadlock detector. Unfortunately, there are underlying difficulties with that solution. First, when a transaction T1 mem-blocks, it depends on all of the transactions holding memory. In particular, it must wait for at least one such transaction to release memory. In ordinary deadlock-detection, however, an edge from T1 to T2 represents that T1 must wait for T2 to complete before continuing. The PostgreSQL deadlock detection code fails to comprehend the dependency involved in a mem-blocked transaction, and our attempts to shoehorn it into the algorithm have failed. Furthermore, the recursive style of the PostgreSQL deadlock cycle detection makes it computationally infeasible to handle extremely dense graphs where most processes are mem-blocked and every process “mem-block-depends” on one another.
Our solution to the heavyweight lock problem involves a post-processing check added to normal deadlock detection. While at first it may appear to be a crude hack, it is indeed the cleanest and most appropriate solution to the problem. When the deadlock detector has checked the system for deadlocked transactions (without using any information about mem-block dependencies), and has resolved any potential cycles, it knows that some process should be able to make progress. At this point, if all transactions are either waiting on a heavyweight lock or are mem-blocked, then there is a deadlock involving a mem-blocked process. Our solution is to simply wake up or abort one of the mem-blocked transactions, and repeat until some process is able to make progress. We refer to a state in which there is at least one process which is not waiting on a lock or memory as there being progress possible.

4.2 Lightweight locks

The second problem above, for mem-blocked transactions holding lightweight locks is much uglier. The problem is that a transaction can acquire a lightweight lock, and then request memory, which forces it to wait indefinitely. If the lightweight lock is used by all other transactions, such as the buffer manager lock, no other transaction cannot make progress. Since PostgreSQL assumes that lightweight locks are held only for a short amount of time, and that usage will never result in deadlocks. Thus, PostgreSQL is not capable of detecting inter-transaction dependencies through lightweight locks. Furthermore, the frequent use of lightweight locks seems to make the increased cost of checking more difficult.

We saw two potential solutions to the lightweight locking problem. The first is to reserve memory for transactions when they acquire lightweight locks. The assumption is that any allocations done while holding the lock will be less than this reserve, and the transaction will never block while holding the lock. Unfortunately, it is not clear how to predict how much memory is needed to be reserved when acquiring the lock. The second solution, which we chose, was to prevent transactions holding lightweight locks from blocking on memory requests. Instead, when the transaction releases all of its lightweight locks, if the transaction has gone over its original allowance, it will block then.

5 Experimental Setup

We examine two memory allocation policies: block-unless-last (BUL) and abort. Each policy monitors the amount of memory used by all transactions in the system, and keeps track as to how much of that memory has been allocated to each. For both policies, we set INITIAL\_ALLOWANCE and GROWTH\_INCREMENT to 64kb.

The BUL policy is the simplest blocking policy which blocks any transaction that makes a request that would put the sum of all allowances over MAX\_ALLOWANCE. The only subtlety is that it will not block the last process not blocked on locks or memory (it “blocks unless it is the last”). Thus, it aims to ensure that the system is always in a state of progress possible. The policy’s who\_to\_force and who\_to\_wakeup decisions are always to wake up (never to abort) an arbitrary mem-blocked transaction.

The abort policy is the simplest aborting policy which aborts any transaction that makes a request that would put the sum of all allowances over MAX\_ALLOWANCE. The policy’s who\_to\_force and who\_to\_wakeup decisions are always to abort an arbitrary mem-blocked transaction.

In our first set of experiments, we consider each policy as we varied MAX\_ALLOWANCE from 100 to 1000 megabytes. For each policy, we run a TPC-C workload with 10 warehouses for 5 minutes. In order to simulate a system under heavy load, where memory pressure would be higher, we use 175 concurrent clients, instead of the TPC-C standard-specified 100 clients. Using the number of New Order transactions completed per minute (TPM-C) as a metric, we evaluate the policies’ performance.

The experiments were all run under PostgreSQL 7.4.2 and Linux 2.4.22 running on a Pentium IV processor running at 1.7GHz, and the system was configured to have 256MB of RAM available. The PostgreSQL shared bufferpool was configured to a conservative 4000 buffers, which seems to maximize system performance. The number is best kept small, as operating system caching duplicates and outperforms the services done by the bufferpool [1].

6 Findings

Figure 1 demonstrates the results of using the BUL policy on the TPC-C system running with 175 clients. The maximum memory allowance MAX\_ALLOWANCE is varied, and the TPM-C for each is plotted. The system was tuned to ensure that a relatively high TPM-C was reached (In fact, we are within 200 TPM-C of the fastest known PostgreSQL TPC-C results when running the baseline system. Additionally, the $/tpmC are 5-10 times better than most entries in the TPC database.).

When the system is limited to too little memory, such as 100MB, the TPM-C is sub-par. The problem here is obviously that transactions are being blocked despite the fact that the system has more than enough available memory (256MB) for them. As the memory allowance is increased, performance increases until the memory allowance starts to pass the amount of physical memory (256MB). The subsequent decrease in performance as MAX\_ALLOWANCE ranges from 256MB to 300MB is due to the fact that the system starts swapping some small amount of data. This interferes significantly with the disk scheduling and causes overall degradation.

Interestingly, as MAX\_ALLOWANCE ranges from 300MB to 750MB, the TPM-C and performance increase. This is contrary to our initial hypothesis, as we expect that increasing MAX\_ALLOWANCE past the available memory would only cause more swapping. We suspect that the reason for the increase is due to the fact that memory allocation is not directly correlated to memory usage. For example, pages within an allocated block may never be used, or may only be used once, and are good candidates for swapping.
Figure 1: System throughput (TPM-C) as a function of maximum memory allowance with the wait-unless-last policy. 64KB growth increment and initial allowance. 175 clients (moderately overloaded), 100 warehouses. Allowance is in Megabytes.

Figure 2: System throughput (TPM-C) as a function of maximum memory allowance with the wait-unless-last policy. 500KB growth increment and initial allowance. 250 clients (very overloaded), 100 warehouses. Allowance is in Megabytes.
out. The virtual memory subsystem is particularly good at identifying these pages and evicting them. As a result, these pages are usable to improve the performance of other DBMS transaction processes.

After increasing \texttt{MAX\_ALLOWANCE} past 750MB, however, we find that the TPM-C begins to decrease. This is consistent with our original hypothesis that increased memory allowance may result in increased swapping. Note that the extreme right point of Figure 1, at 1000MB allowance, is equivalent to the standard PostgreSQL memory allocator with no restrictions, since the amount is greater than the system virtual and physical memory.

Figure 2 depicts the results of an experiment similar to those in Figure 1. The differences are that the system load was increased to 250 clients to increase memory pressure, and the initial allowance and growth increment parameters were set larger, to 500K. This reflects the suspicion detailed earlier in which extremely low growth increments and initial allowances may cause performance loss, which are more significant in a system with heavier loads. As can be seen from the figure, the data appears to exhibit a similar trend to that of Figure 1. The performance degrades as the system starts to use more than real RAM, up to a point. From there, performance increases, as the virtual memory system removes unnecessary cold pages and increases the effective multiprogramming level. Finally, performance degrades as the RAM pressure increases enough. Note that the performance of the system with heavier load is, in general, better than the performance of the system with lighter load. We believe this is due to the fact that using 500K growth and initial allowance parameters reduces the algorithm’s overhead.

Our efforts with the \texttt{abort} policy have exhibited significant issues relative to the \texttt{BUL} policy. The initial per-process memory allocation and the growth increment are fairly low, at 64K. The result is that transactions acquire memory in piecemeal, and have a very high probability of memory contending on one another. In this circumstance, memory conflicts are so large and aborts are so prevalent that the TPM-C drops to 18, since most transactions die.

One possible solution to make \texttt{abort} a better policy is to increase the initial memory allowance and the growth increment. We find that most transactions need only one or two megabytes of data. As a result, we increase these parameters both to 500K, and reevaluate the \texttt{abort} policy. In this case, the TPM-C increases to 58 in the best case. Unfortunately, this is still much lower than the TPM-C that we can get using the \texttt{BUL} policy, or no memory control at all.

Figure 3 shows the performance of the \texttt{BUL} policy relative to the standard PostgreSQL, as a function of the number of clients in the system. The plot depicts the TPM-C for \texttt{BUL} with a 500MB and a 750MB \texttt{MAX\_ALLOWANCE}. This data is synthesized from the data shown in Figures 1 and 2. While under extremely heavy loads, a limit of 500MB is best, limiting the system to 750MB seems to be the most effective solution for these experiments. Extensive analysis must be done to determine how dependent \texttt{BUL}’s performance is on the number of clients in the system.

7 Conclusions

We’ve implemented a centralized memory management system as described above, and evaluated its effectiveness at improving throughput for TPC-C workloads under heavy load.

First, we considered a moderate amount of overload, and relatively small growth and initial allowance parameters, which pessimistically biases our results. Despite this bias, we find that throughput is improved around 10% by restricting memory allocation using the \texttt{BUL} policy. We also predict that it is apparently valuable to overcommit memory, and rely on the effectiveness of a virtual memory subsystem to evict unused or cold heap memory pages. Second, we considered a more heavily loaded system, and increased the growth and initial allowance parameters, in which our algorithms should perform better. We find that restricting memory usage does indeed improve overall performance by about 40% in this case. Determining where the optimal performance will be reached, however, does not appear to be an easy problem to solve.

We find that our simple abort policies are ineffective, because of extra overheads introduced in restarting transactions.

We have not yet evaluated the effect of the overhead of maintaining the data structures used by the memory manager. The code is simple, and we’ve taken some effort to avoid overhead when possible, but there is locking involved.

8 Response To Feedback

We recieved three primary issues for feedback for our project. The first is that our system was tuned improperly, and that our results were inconclusive. The second is that the interpretation of our graphs is likely correct, but needs in-depth experimental evidence to back it up. The third is that it would be interesting to compare the performance of the normal PostgreSQL system to ours as the number of clients and system load increases.

Sadly, the bulk of our work since the presentation has gone exclusively into retuning the database, and ensuring that the TPM-C performance is reasonable. After thorough investigation, we have come to the conclusion that Natassa’s expectations of PostgreSQL’s performance are fairly unreasonable. While a system such as Oracle or DB2 can achieve several hundred TPM-C on a multiprocessor with a RAID array, it appears completely unreasonable for TPM-C on a two-disk single processor machine. The suspicion that Bianca has achieved 900 TPM-C appears to be completely unsubstantiated. Apparently, the best published TPM-C results for PostgreSQL on a singleprocessor machine is around 300 TPM-C, which we get fairly close to (100-200 TPM-C). For our hardware and configuration, these numbers are conclusively correct.
Figure 3: Performance of BUL using 500MB Max Allowance (MemMgr(500)) and 750MB Max Allowance (MemMgr(750)), relative to standard PostgreSQL, as a function of the number of clients in the system.

After some careful thought and evaluation, it is pretty unrealistic to greatly improve the statistical analysis for Figures 1 and 2. The problem is that the trends are dependent on the actual costs incurred each time that the system encounters a page fault. Under Linux, it is impossible to directly measure and evaluate those costs. An in-depth statistical analysis will require two to three weeks to hack the kernel to properly account the necessary statistics.

References