PostgreSQL Memory Management Retrofit

Benoît Hudson and David McWerther

April 1, 2004

1 Group Info

Benoît Hudson (bhudson@cs.cmu.edu)
David McWerther (cache@cs.cmu.edu)

Web page: http://www.cs.cmu.edu/~bhudson/projects/dbase

2 Project Description

Memory resources in PostgreSQL, in general, are poorly managed. A set of DBMS processes allocate and use portions of RAM, independently of the memory needs of the other processes. As a result, under heavy system loads, the OS may be required to overcommit memory, leading to pathetically poor performance due to virtual memory swapping. Even worse, since the OS has no knowledge about the behavior and needs of the DBMS, its virtual memory swapping policies may result in further performance degradation.

First, we make the observation that in general, swapping out DBMS memory can cause a large performance hit. In particular, each swap means that an operation that might have been all in memory now has two I/O operations: one write to disk and one read. One possible solution is to forbid swapping out DBMS memory, by refusing to overcommit the physical memory available to the DBMS. This may or may not be the optimal solution, however, as some swapping may be able to improve performance by temporarily giving critical algorithms additional memory. We will implement a centralized memory management scheme in PostgreSQL that will enable us to vary the amount of overcommitted memory and evaluate the performance of the system as this parameter is varied.

Second, once a centralized memory management controller is in place, it is essential to determine what to do when not enough memory is available to a transaction. We will evaluate two policies: suspend, abort. The suspend policy simply blocks the current holder, and temporarily considers its current memory as unallocated until enough memory is available to complete the request (thus its memory will be overcommitted
The abort policy aborts the transaction currently making the request (freeing its memory) and uses admission control to prevent it from immediately reentering the system until sufficient memory is available.

Third, we recognize that certain database functions have different memory requirements, and are affected by page faults in different ways. Certain operations’ memory, such as sort buffers, appear to be much more vulnerable to page-faults than other operations’, such as hash tables. Assuming $N$ data elements and a uniform distribution of page faults, a sort that makes $O(N\lg N)$ memory accesses may suffer much more than a hash operation that makes $O(N)$ accesses. While considering overcommit memory allocations, we will examine the benefits of locking certain (sort) buffers in memory, to ensure decent performance.

One additional consideration is that we want to avoid deadlocks: transaction $A$ could be locked waiting for transaction $B$ to release a synchronization lock, while transaction $B$ is blocked waiting for more memory to become available. We can sidestep this question by using the same locking mechanism for memory allocation as for synchronization.

We will compare the performance of our various policies against an unmodified postgres on several benchmarks. Interesting values include: transactions per second, mean response time of transactions, and the number of page faults (that is, the utilization of virtual memory), each plotted against the number of transactions. We want to show that in all cases, the overhead of our technique is small; and that in cases of heavy memory load, our performance degrades gracefully.

Summary:

- Admission control: don’t let processes start if we’re low on memory.
- Overcommit policy: if we run out of memory, either suspend processes or abort them.
- Scheduling: when we have enough memory again, which process do we let in or unsuspend? First-come first-served may not be optimal; we may need additional considerations (allow the longest-running process? the one holding the most locks?)
- Memory tagging: Some memory causes more pain if it’s swapped out than other memory. We think sorting is painful to swap, so tag that memory as illegal for the OS to swap out.

### 3 Logistics

#### 3.1 Schedule

Week 1-2: Design and architecture and implementation of central memory administrator.
Week 3-4: Implementation of overcommit/no-overcommit and suspend/abort policies.
3.2 Milestone

We will have implemented a centralized memory allocation service for PostgreSQL, including facilities for considering the experimental options of:

- Overcommit amount (how much to overcommit memory)
- Abort vs Suspend overcommit avoidance policy
- Locking of critical buffers into physical memory

After the milestone, we will concentrate on evaluation of the system using a TPC-C workload. If necessary, synthetic microbenchmarks will also be developed to more thoroughly evaluate system performance.

3.3 Related Work

The concept of explicit memory management has been dealt with in countless contexts. In general, however, the problem is quite difficult in the transactional DBMS setting. Our implementation is novel in the research community in the sense that it retrofits an existing system, which has poorly designed its memory management subsystem, to perform better.

3.4 Resources Needed

We will exploit the existing PostgreSQL source code, along with an implementation of the TPC-C workload for PostgreSQL. All of the software is currently acquired. Experimentation will be executed on commodity research computing hardware already available to the experimentors.

3.5 Getting Started

We have currently established the architecture and design of the implementation of centralized memory management as well as implementation of each of the experimental policy options. The only undesigned aspect is whether and how to predict memory usage for admission control.