Inside the PostgreSQL Shared Buffer Cache

Greg Smith

Truviso

07/07/2008

About this presentation

- The master source for these slides is

  `http://www.westnet.com/~gsmith/content/postgresql`

- You can also find a machine-usable version of the source code to the later internals sample queries there
- This presentation comes from research done while rewriting the background writer for PostgreSQL 8.3
- There's very little information about PostgreSQL buffer cache internals available anywhere outside of the source code

Database organization

- Databases are mainly a series of tables
- Each table gets a subdirectory
- In that directory are a number of files
- A single files holds up to 1GB of data (staying well below the 32-bit 2GB size limit)
- The file is treated as a series of 8K blocks

Buffer cache organization

- shared_buffers sets the size of the cache (internally, NBuffers)
- The buffer cache is a simple array of that size
- Each cache entry points to an 8KB block (sometimes called a page) of data
- In many scanning cases the cache is as a circular buffer; when NBuffers is reached things scanning the buffer cache start over at 0
- Initially all the buffers in the cache are marked as free

Entries in the cache

- Each buffer entry has a tag
- The tag says what file (and therefore table) this entry is buffering and which block of that file it contains
- A series of flags show what state this block of data is in
- Pinned buffers are locked by a process and can't be used for anything else until that's done.
- Dirty buffers have been modified since they were read from disk
- There are several other flags and fields such as locks used for internal purposes; you can ignore those.
- The usage count estimates how popular this page has been recently

Buffer Allocation

- When a process wants a buffer, it calls BufferAlloc with what file/block it needs
- If the block is already in the cache, it gets pinned and then returned
- Otherwise, a new buffer must be found to hold this data
- If there are no buffers free (there usually aren't) BufferAlloc selects a buffer to evict to make space for the new one
- If that page is dirty, it is written out to disk. This can cause the backend trying to allocate that buffer to block as it waits for that write I/O to complete.
- The block on disk is read into the page in memory
- Pages all start out pinned until the process that requested the data releases (unpins) them.
- The pinning process, whether the buffer was found initialy or it had to be read from disk, also increases the usage_count of the buffer. This is the only way that count increases.

Basic Buffer Eviction Theory

- Deciding which entry should be removed from a cache to allocate a new one is a classic computer science problem
- The usual first implementation people build is Least Recently Used (LRU): always evict the buffer that has gone the longest since it was last used for something.
- One implementation stores the timestamp when the page was last used in order to determine the LRU bufer
- Another way keeps pages sorted in order of recent access
- Both of these "perfect LRU" approaches are complicated to build
- A simple LRU has no memory, so a page that been accessed many times in the past and then not for a while is indistinguishable from one that was accessed once only at that same time.

From LRU to Clock Sweep

- Sequential scans of things bigger than memory are an example of a problem area for a pure LRU. Such a scan will replace everything else in the cache with data that is only needed once. These are very common in database applications.
- A design that doesn't have this problem is "scan resistant"
- The simplest such design is a "second chance" algorithm, where each time a page is referenced it is marked as unsuitable for eviction. When page that is up for eviction, the reference is cleared and it goes to the back of the list of eviction candiates.
- A way to implement this is the "clock sweep", where the buffer cache is treated as a circular list. Each time the eviction scanner gets to the end it starts back over at the beginning again.

Improving eviction with usage counts

- To further sort popular pages that should be kept in memory from ones that are safer to evict, the usage count that's incremented by BufferAlloc is used
- The eviction "Strategy" continuously does a circular scan of the buffer cache
- Any page that has a non-zero usage count is safe from eviction
- Pinned pages obviously can't be evicted either
- When the Strategy considers a buffer, if it's not evicted the usage count is decreased
- The maximum usage count any buffer can get is set by BM_MAX_USAGE_COUNT, currently fixed at 5
- This means that a popular page that has reached usage_count=5 will survive 5 passes over the entire buffer cache before it's possible to evict it.

Optimizations for problem areas

- Version 8.3 adds some "buffer ring" features to lower buffers used by activity like sequential scans and VACUUM.
- This re-uses a subset of the buffers continuously rather than allocating and evicting the way everything else does
- If you are doing a VACUUM or a scan where the table is larger than (shared buffers/4), you get a ring list to keep track of requested buffers
- The ring size is normally 256K, as you request pages they get added to the list of ones in the ring
- Once the ring is full and you circle around to a page that's already been used, if nobody else has used it since it was put in there (usage count=0,1) that page is evicted and re-used rather than allocating a new one
- If someone else is using the buffer, a new one is allocated normally, replacing the original ring entry

Monitoring buffer activity with pg_stat_bgwriter

- A new view in 8.3 allows tracking gross statistics about how things move in and out of the buffer cache
- select * from pg_stat_bgwriter
- buffer_alloc is the total number of calls to allocate a new buffer for a page (whether or not it was already cached)
- buffers_backend says how may buffer clients and other backends had to write to fulfill those allocations, which can cause a wait for I/O that disrupts them
- buffers_clean tells you how many buffers the background writer cleaned for you in advance of that
- maxwritten_clean tells you if the background writer isn't being allowed to work hard enough
- buffers_checkpoint gives a count of how many buffers the checkpoint process had to write for you
- Comparing checkpoints_timed and checkpoints_req shows whether you've set checkpoint_segments usefully

Interaction with the Operating System cache

- PostgreSQL is designed to rely heavily on the operating system cache, because portable sotware like PostgreSQL can't know enough about the filesystem or disk layout to make optimal decisions about how to read and write files
- The shared buffer cache is really duplicating what the operating system is already doing: caching popular file blocks
- In many cases, you'll find exactly the same blocks cached by both the buffer cache and the OS page cache
- This makes is a bad idea to give PostgreSQL too much memory to manage
- But you don't want to give it too little because the OS is probably using a simpler LRU scheme rather than a database optimized clock-sweep approach
- Recent research suggests you can spy on the OS if it supports the right APIs; see `http://www.kennygorman.com/wordpress/?p=250`

Recommended shared_buffers sizing

- With a modern system where there's typically 2GB or more of total RAM, anecdotal testing suggests a dedicated database server should get 1/4 to 1/3 of total RAM
- Smaller memory configurations need to be more careful about considering overhead for the OS, database, and applications
- The remaining RAM after OS+database+applications should be used for disk caching, and effective_cache_size should be updated with a useful estimate of that amount
- Larger systems with more static, read-only applications might utilize a higher percentage
- Reports on the point of diminishing returns where increasing shared_buffers further is ineffective in PostgreSQL 8.3 are usually in the range of 2 to 10GB of dedicated RAM

Exceptions to sizing rules

- Some applications might prefer having more work_mem for sorting instead
- Windows platforms do not perform well with large shared_buffers settings. The effective maximum size is somewhere around 10,000-50,000 buffers even if you have much more RAM than that available.
- This is why I recommended this talk was more appropriate for UNIX-like platforms, you can't really take full advantage of the buffer cache on Windows
- Older PostgreSQL versions (before 8.1) did not use the current clock-sweep algorithm. They also had locking issues that kept managing large amounts of memory here problematic. Around 10,000 is as large as you want to go in 8.0 or earlier.
- 8.1 removed the single BufMgrLock that was another bottleneck on having too much happen in shared_buffers. In 8.2 that was further split into NUM_BUFFER_PARTITIONS

Checkpoint considerations

- Systems doing heavy amounts of write activity can discover checkpoints are a serious problem
- Checkpoint spikes can last several seconds and essentially freeze the system.
- The potential size of these spikes go up as the memory in shared_buffers increases.
- There is a good solution for this in 8.3 called checkpoint_completion_target, but in 8.2 and before it's hard to work around.
- Since only memory in shared_buffers participates in the checkpoint, if you reduce that and rely on the OS disk cache instead, the checkpoint spikes will reduce as well.

Increase or decrease shared_buffers?

- Since there are different trade-offs in either direction, how can you tell whether your current buffer cache is too small or too big?
- You want to balance the space used by the smarter clock-sweep algorithm against the larger OS cache, and perhaps get some benefit from synergy between the two different approaches
- Traditional approach is to look at system stats tables like pg_stat_all_tables, pg_stat_all_index, pg_statio_all_tables, etc. to note hit percentages
- If you also collect data from inside the buffer cache, you can improve estimation here by comparing the amount of a table or index that is cached by the database with the size on disk
- Don't forget that there's normally nothing tracking what the OS is caching for you

Looking inside the buffer cache: pg_buffercache

```
cd contrib/pg_buffercache
make
make install
psql -d database -f pg_buffercache.sql
```

- You can take a look into the shared_buffer cache using the pg_buffercache module
- The 8.3 version includes the usage_count information, earlier versions did not. The patch to add usage count is simple and applies easily to 8.2 and possibly earlier versions. See `http://archives.postgresql.org/pgsql-patches/2007-03/ms`
- for the patch, apply before doing the above

Limitations of pg_buffercache

- While the cache is shared among the entire cluster, the module gets installed into one database and can only decode table names in that database
- The select that grabs this information takes many locks inside the database and is very disruptive to activity
- As Murphy predicts, the time you'd most like to collect this information (really busy periods) is also the worst time to run this inspection
- While it can be helpful to collect this information regularly via cron or pgagent, frequent snapshots will impact system load.
- Consider caching the information into a temporary table if you're doing more than one pass over it

Simple pg_buffercache queries: Top 10

```
SELECT c.relname,count(*) AS buffers
FROM pg_class c INNER JOIN pg_buffercache b
ON b.relfilenode=c.relfilenode INNER JOIN pg_database d
ON (b.reldatabase=d.oid AND d.datname=current_database())
GROUP BY c.relname
ORDER BY 2 DESC LIMIT 10;
```

- You must join against pg_class to decode the file this buffer is caching a block from
- Top 10 tables represented in the cache and how much memory they have:
- Remember: we only have the information to decode tables in the current database

Sample internals output

- To show some samples, we start by using pgbench to generate some simulated activity
- Running 'pgbench -S' does a simple benchmark that runs SELECT on an accounts table with an account number as its primary and only key
- The ability to cache most of the index information on a popular table is often the key to good database performance
- We size the database so it just barely fits in RAM, but not the shared_buffers cache
- The queries to generate these more complicated samples are too large for these slides; see the web page referenced at the beginning for their source code

Creating a pgbench database and running a test - 2GB of RAM in server

```
# pgbench -i -s 100 pgbench
# pg_ctl stop ; pg_ctl start
select pg_size_pretty(pg_database_size('pgbench'));
  1416 MB
select pg_size_pretty(pg_relation_size('accounts'));
  1240 MB
select pg_size_pretty(pg_relation_size('accounts_pkey'));
  171 MB
show shared_buffers
  60000 (469MB)
# pgbench -S -c 8 -t 10000 pgbench
```

Buffer contents summary

```
relname |buffered| buffers % | % of rel
accounts | 306 MB | 65.3 | 24.7
accounts_pkey | 160 MB | 34.1 | 93.2

usagecount | count | isdirty
0 | 12423 | f
1 | 31318 | f
2 | 7936 | f
3 | 4113 | f
4 | 2333 | f
5 | 1877 | f
```

Usage count summary and the background writer

```
select usagecount,count(*),isdirty from pg_buffercache
group by isdirty,usagecount order by isdirty,usagecount;
```

- This gives you an idea the balance of popular pages versus probably transient ones
- Useful for tuning the pre-8.3 background writer (if you've applied the usage count patch)
- The LRU scan will only write pages with a usage count of 0, this tells you if that will be effective
- If you have many pages that get dirty with a usage count of 4 or 5, tuning the all scan too aggressively will result in lots of wasted I/O as the popular pages get written repeatedly
- High counts of dirty pages means the next checkpoint will be a big one

Usage count distribution

```
relname | buffers | usage
accounts | 10223 | 0
accounts | 25910 | 1
accounts | 2825 | 2
accounts | 214 | 3
accounts | 14 | 4
accounts_pkey | 2173 | 0
accounts_pkey | 5392 | 1
accounts_pkey | 5086 | 2
accounts_pkey | 3747 | 3
accounts_pkey | 2296 | 4
accounts_pkey | 1756 | 5
```

Integrating usage counts into resizing: decreasing size

- The balance of popular (high usage count) versus transient (low usage count) pages tells you a lot about whether your cache is sized appropriately
- If most of your pages have low usage counts (0,1), but you're still getting good hit rates, you can probably decrease the size of the buffer cache.
- Even the simplest operating system LRU algorithm is capable of usefully caching in situations where there aren't popular pages to prioritize.
- A smaller buffer cache means shorter checkpoints and might speed up allocations of new pages. Usage counts can't get as high if you're constantly passing over the pages with the clock sweep.

Integrating usage counts: increasing size

- If a high percentage of the pages have high usage count (4,5), you likely aren't caching as many popular pages as you should and the buffer cache would benefit from being increased in size.
- You need to carefully consider how many of those pages are dirty though before doing that, as lots of dirty and popular pages means big checkpoints (especially before 8.3)
- This situation is less clear than the low count case
- If the cache is too small, it will be hard for any usage counts to grow large when all pages are constantly being passed over by the clock sweep hand.
- Be sure to combine this with a look at the traditional stat hit percentages
- It's very useful if you can put together a benchmark representative of your application to quantify whether an increase helped.