In and Out of the PostgreSQL Shared Buffer Cache

Greg Smith

2ndQuadrant US

05/21/2010

About this presentation

- The master source for these slides is

  `http://projects.2ndquadrant.com`

- You can also find a machine-usable version of the source code to the later internals sample queries there

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

2ndQuadrant
Professional PostgreSQL

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 all buffers are used, 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
- The usage count estimates how popular this page has been recently
- Good read cache statistics available in views like pg_statio_user_tables

2ndQuadrant
Professional PostgreSQL

Buffer Allocation

- When a process wants to access a block, it requests a buffer allocation for it
- If the block is already cached, its returned with increased usage count
- Otherwise, a new buffer must be found to hold this data
- If there are no buffers free (there usually aren't) a buffer is evicted to make space for the new one
- If that page is dirty, it is written out to disk, and the backend waits for that write
- The block on disk is read into the page in memory
- The usage count of an allocated buffer starts at 1

2ndQuadrant Professional PostgreSQL

Eviction with usage counts

- The usage count is used to sort popular pages that should be kept in memory from ones that are safer to evict
- Buffers are scanned sequentially, decreasing their usage counts the whole time
- Any page that has a non-zero usage count is safe from eviction
- 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.

2ndQuadrant Professional PostgreSQL

Interaction with the Operating System cache

- PostgreSQL is designed to rely heavily on the operating system cache
- The shared buffer cache is really duplicating what the operating system is already doing: caching popular file blocks
- Exactly the same blocks can be cached by both the buffer cache and the OS page cache
- It's a bad idea to give PostgreSQL too much memory
- But you don't want to give it too little. The OS is probably using a LRU scheme, not a database optimized clock-sweep
- You can spy on the OS cache using pg_fincore

Looking inside the buffer cache: pg_buffercache

- You can take a look into the shared_buffer cache using the pg_buffercache module
- 8.3 and later versions includes the usage_count information

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

2ndQuadrant
Professional PostgreSQL

Limitations of pg_buffercache

- Module is installed into one database, can only decode table names in that database
- Viewing the data takes many locks inside the database, very disruptive
- When you'd most like to collect this information is also the worst time to do this expensive query
- Frequent snapshots will impact system load, might collect occasionallly via cron or pgagent, .
- Cache the information if making 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;
```

- Join against pg_class to decode the file this buffer is caching
- Top 10 tables in the cache and how much memory they have
- Remember: we only have the information to decode tables in the current database

2ndQuadrant
Professional PostgreSQL

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
```

2ndQuadrant Professional PostgreSQL

General shared_buffers sizing rules

- Anecdotal tests suggest 15% to 40% of total RAM works well
- Start at 25% and tune from there
- 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.
- Only memory in shared_buffers participates in the checkpoint
- Reduce that and rely on the OS disk cache instead, the checkpoint spikes will reduce as well.

2ndQuadrant
Professional PostgreSQL

Monitoring buffer activity with pg_stat_bgwriter

- select * from pg_stat_bgwriter - added in 8.3
- Statistics about things moving in and out of the buffer cache
- Need to save multiple snapshots with a timestamp on each to be really useful
- buffer_alloc is the total number of calls to allocate a new buffer for a page (whether or not it was already cached)
- Comparing checkpoints_timed and checkpoints_req shows whether you've set checkpoint_segments usefully

2ndQuadrant
Professional PostgreSQL

Three ways for a buffer to be written

- buffers_checkpoint: checkpoint reconciliation wrote the buffer
- buffers_backend: client backend had to write to satisfy an allocation
- buffers_clean: background writer cleaned a dirty buffer expecting an allocation
- maxwritten_clean: The background writer isn't being allowed to work hard enough

Derived statistics

- Timed checkpoint %
- % of buffers written by checkpoints, background writer cleaner, backends
- If you have two snapshots with a time delta, can compute figures in real-world units
- Average minutes between checkpoints
- Average amount written per checkpoint
- Buffer allocations per second * buffer size / interval = buffer allocations in MB/s
- Total writes per second * buffer size / interval = avg buffer writes in MB/s

2ndQuadrant
Professional PostgreSQL

Spreadsheet

- Sometimes slides are not what you want

2ndQuadrant
Professional PostgreSQL

Iterative tuning with pg_buffercache and pg_stat_bgwriter

- Increase checkpoint_segments until time between is reasonable
- Increase shared_buffers until proportion of high usage count buffers stop changing
- Positive changes should have a new MB/s write figure and changed checkpoint statistics
- Optimize system toward more checkpoint writes, and total writes should drop
- Go too far and the size of any one checkpoint may be uncomfortably large, causing I/O spikes
- When performance stops improving, you've reached the limits of usefully tuning in this area

2ndQuadrant
Professional PostgreSQL

Wrap-up

- Database buffer cache is possible to instrument usefully
- Saving regular usage snapshots allows tracking internals trends
- It's possible to measure the trade-offs made as you adjust buffer cache and checkpoint parameters
- No one tuning is optimal for everyone, workloads have very different usage count profiles

2ndQuadrant Professional PostgreSQL

Credits

- Contributors toward the database statistics snapshots in the spreadsheet:
- Kevin Grittner (Wisconsin Courts)
- Ben Chobot (Silent Media)

2ndQuadrant
Professional PostgreSQL

Questions?

- The "cool" kids hang out on pgsql-performance