PostgreSQL Hardware Performance Tuning

Bruce Momjian

17th April 2002

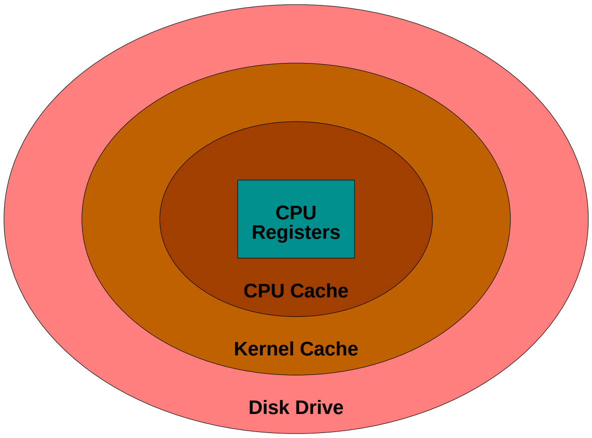
POSTGRESQL is an object-relational database developed on the Internet by a group of developers spread across the globe. It is an open-source alternative to commercial databases like Oracle and Informix.

POSTGRESQL was originally developed at the University of California at Berkeley. In 1996, a group began development of the database on the Internet. They use email to share ideas and file servers to share code. POSTGRESQL is now comparable to commercial databases in terms of features, performance, and reliability. It has transactions, views, stored procedures, and referential integrity constraints. It supports a large number of programming interfaces, including ODBC, Java (JDBC), TCL/TK, PHP, Perl, and Python. POSTGRESQL continues to improve at a tremendous pace thanks to a talented pool of Internet developers.

Performance Concepts

There are two aspects of database performance tuning. One is improving the database's use of the CPU, memory, and disk drives in the computer. The second is optimizing the queries sent to the database. This article talks about the hardware aspects of performance tuning. The optimization of queries is done using SQL commands like CREATE INDEX, VACUUM, VACUUM ANALYZE, CLUSTER, and EXPLAIN. These are discussed in my book, *PostgreSQL: Introduction and Concepts* at `http://www.postgresql.org/docs/awbook.html`.

To understand hardware performance issues, it is important to understand what is happening inside the computer. For simplicity, a computer can be thought of as a central processing unit (CPU) surrounded by storage. On the same chip with the CPU are several CPU registers which store intermediate results and various pointers and counters. Surrounding this is the CPU cache which holds the most recently accessed information. Beyond the CPU cache is a large amount of random-access main memory (RAM) which holds executing programs and data. Beyond this main memory are disk drives, which store even larger amounts of information. Disk drives are the only permanent storage area, so anything to be kept when the computer is turned off must be placed there. In summary, here are the storage areas surrounding the CPU:



Storage Area	Measured in
CPU registers	bytes
CPU cache	kilobytes
RAM	megabytes
disk drives	gigabytes

You can see that storage areas increase in size as they get farther from the CPU. Ideally, a huge amount of permanent memory could be placed right next to the CPU, but this would be too slow and expensive. In practice, the most frequently used information is stored next to the CPU, and less frequently accessed information is stored farther away and brought to the CPU as needed.

Keeping Information Near the CPU

Moving information between various storage areas happens automatically. Compilers determine which information should be stored in registers. CPU chip logic keeps recently used information in the CPU cache. The operating system controls which information is stored in RAM and shuttles it back and forth from the disk drive.

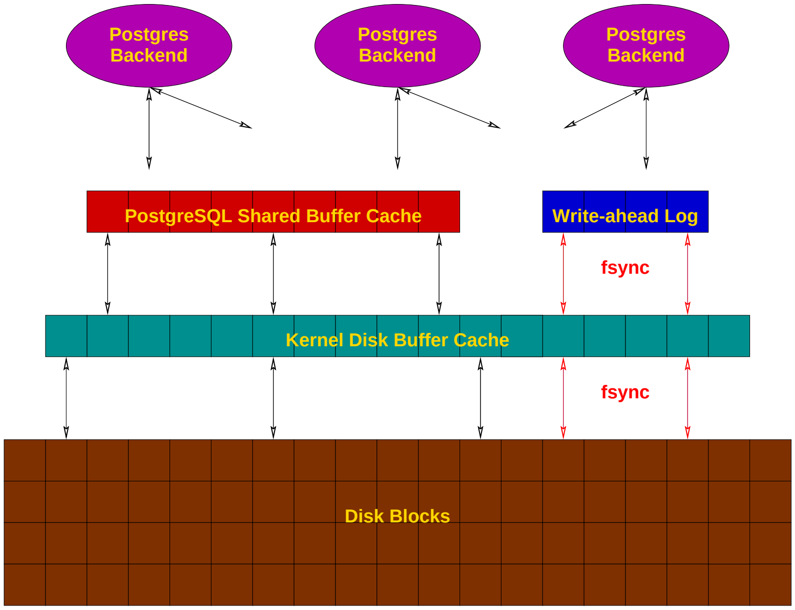
CPU registers and the CPU cache cannot be effectively tuned by the database administrator. Effective database tuning involves increasing the amount of useful information in RAM, thus preventing disk access where possible.

You might think this is easy to do, but it is not. A computer's RAM contains many things:

- executing programs
- program data and stack
- POSTGRESQL shared buffer cache
- kernel disk buffer cache
- kernel

Proper tuning involves keeping as much database information in RAM as possible while not adversely affecting other areas of the operating system.

POSTGRESQL Shared Buffer Cache

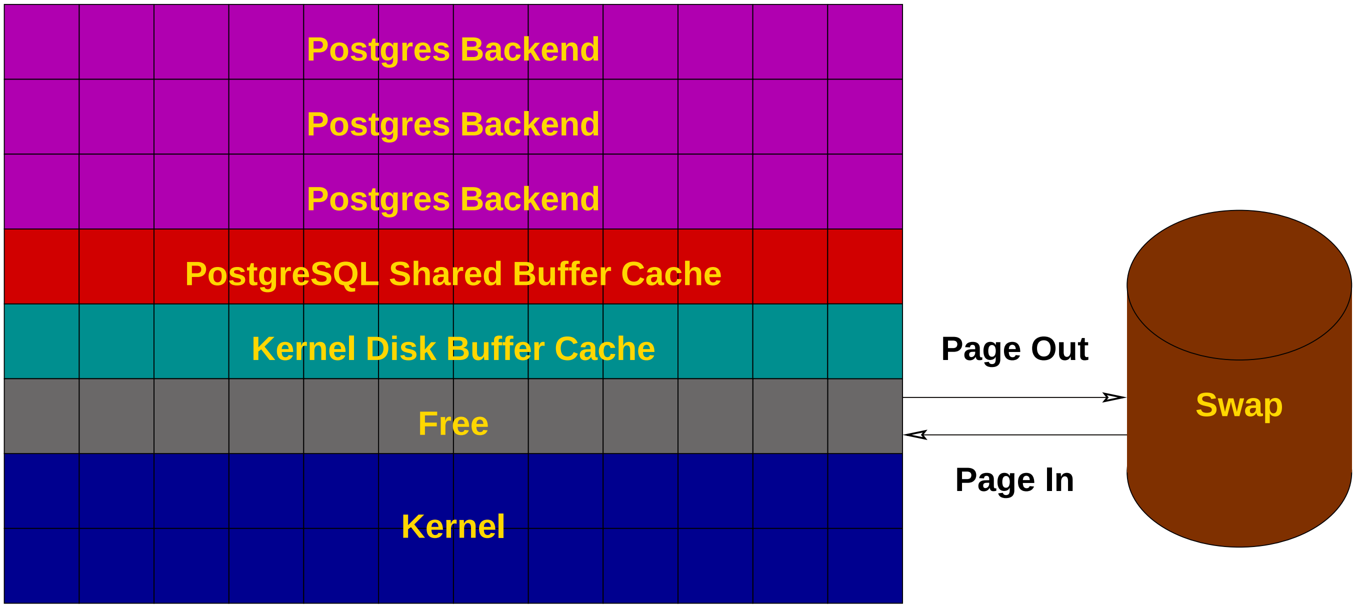


POSTGRESQL does not directly change information on disk. Instead, it requests data be read into the POSTGRESQL shared buffer cache. POSTGRESQL backends then read/write these blocks, and finally flush them back to disk.

Backends that need to access tables first look for needed blocks in this cache. If they are already there, they can continue processing right away. If not, an operating system request is made to load the blocks. The blocks are loaded either from the kernel disk buffer cache, or from disk. These can be expensive operations.

The default POSTGRESQL configuration allocates 64 shared buffers. Each buffer is 8 kilobytes. Increasing the number of buffers makes it more likely backends will find the information they need in the cache, thus avoiding an expensive operating system request. The change can be made with a *postmaster* command-line flag or by changing the value of *shared_buffers* in *postgresql.conf*.

How Big Is Too Big?



You may think, "I will just give all my RAM to the POSTGRESQL shared buffer cache." However, if you do that, there will be no room for the kernel or for any programs to run. The proper size for the POSTGRESQL shared buffer cache is the largest useful size that does not adversely affect other activity.

To understand adverse activity, you need to understand how UNIX operating systems manage memory. If there is enough memory to hold all programs and data, little memory management is required. However, if everything doesn't fit in RAM, the kernel starts forcing memory pages to a disk area called *swap*. It moves pages that have not been used recently. This operation is called a *swap pageout*. Pageouts are not a problem because they happen during periods of inactivity. What is bad is when these pages have to be brought back in from swap, meaning an old page that was moved out to swap has to be moved back into RAM. This is called a *swap pagein*. This is bad because while the page is moved from swap, the program is suspended until the pagein completes.

Pagein activity is shown by system analysis tools like *vmstat* and *sar* and indicates there is not enough memory available to function efficiently. Do not confuse swap pageins with ordinary pageins, which may include pages read from the filesystem as part of normal system operation. If you can't find swap pageins, many pageouts is a good indicator you are are also doing swap pageins.

Effects of Cache Size

You may wonder why cache size is so important. First, imagine the POSTGRESQL shared buffer cache is large enough to hold an entire table. Repeated sequential scans of the table will require no disk access because all the data is already in the cache. Now imagine the cache is one block smaller than the table. A sequential scan of the table will load all table blocks into the cache until the last one. When that block is needed, the oldest block is removed, which in this case is the first block of the table. When another sequential scan happens, the first block is no longer in the cache, and to load it in, the oldest block is removed, which in this case is now the second block in the table. This pushing out of the next needed block continues to the end of the table. This is an extreme example, but you can see that a decrease of one block can change the efficiency of the cache from 100% to 0%. It shows that finding the right cache size can dramatically affect performance.

Proper Sizing of Shared Buffer Cache

Ideally, the POSTGRESQL shared buffer cache will be:

- Large enough to hold most commonly-accessed tables
- Small enough to avoid *swap pagein* activity

Keep in mind that the *postmaster* allocates all shared memory when it starts. This area stays the same size even if no one is accessing the database. Some operating systems pageout unreferenced shared memory, while others lock shared memory into RAM. Locked shared memory is preferred. The POSTGRESQL administrators guide has information about kernel configuration for various operating systems, `http://developer.postgresql.org/docs/postgres/kernel-resources.html`

Sort Memory Batch Size

Another tuning parameter is the amount of memory used for sort batches. When sorting large tables or results, POSTGRESQL will sort them in parts, placing intermediate results in temporary files. These files are then merged and resorted until all rows are sorted. Increasing the batch size creates fewer temporary files and often allows faster sorting. However, if the sort batches are too large, they cause pageins because parts of the sort batch get paged out to swap during sorting. In these cases, it is much faster to use smaller sort batches and more temporary files, so again, swap pageins determine when too much memory has been allocated. Keep in mind this parameter is used for every backend performing a sort, either for ORDER BY, CREATE INDEX, or for a *merge join*. Several simultaneous sorts will use several times this amount of memory.

This value can be changed using a *postmaster* command-line flag or by changing the value of *sort_mem* in *postgresql.conf*.

Cache Size and Sort Size

Both cache size and sort size affect memory usage, so you cannot maximize one without affecting the other. Keep in mind that cache size is allocated on *postmaster* startup, while sort size varies depending on the number of sorts being performed. Generally, cache size is more significant than sort size. However, certain queries that use ORDER BY, CREATE INDEX, or merge joins may see speedups with larger sort batch sizes.

Also, many operating systems limit how much shared memory can be allocated. Increasing this limit requires operating system-specific knowledge to either recompile or reconfigure the kernel. More information can be found in the POSTGRESQL administrators guide, `http://developer.postgresql.org/docs/postgres/kernel-resources.html`.

As a start for tuning, use 25% of RAM for cache size, and 2-4% for sort size. Increase if no swapping, and decrease to prevent swapping. Of course, if the frequently accessed tables already fit in the cache, continuing to increase the cache size no longer dramatically improves performance.

Disk Locality

The physical nature of disk drives makes their performance characteristics different from the other storage areas mentioned in this article. The other storage areas can access any byte with equal speed. Disk drives, with their spinning platters and moving heads, access data near the head's current position much faster than data farther away.

Moving the disk head to another cylinder on the platter takes quite a bit of time. Unix kernel developers know this. When storing a large file on disk, they try to place the pieces of the file near each other. For example, suppose a file requires ten blocks on disk. The operating system may place blocks 1-5 on one cylinder and blocks 6-10 on another cylinder. If the file is read from beginning to end, only two head movements are required — one to get to the cylinder holding blocks 1-5, and another to get to blocks 6-10. However, if the file is read non-sequentially, e.g. blocks 1,6,2,7,3,8,4,9,5,10; ten head movements are required. As you can see, with disks, sequential access is much faster than random access. This is why POSTGRESQL prefers sequential scans to index scans if a significant portion of the table needs to be read. This also highlights the value of the cache.

Multiple Disk Spindles

The disk head moves around quite a bit during database activity. If too many read/write requests are made, the drive can become saturated, causing poor performance. (Vmstat and sar can provide information on the amount of activity on each disk drive.)

One solution to disk saturation is to move some of the POSTGRESQL data files to other disks. Remember, moving the files to other filesystems on the same disk drive does not help. All filesystems on a drive use the same disk heads.

Database access can be spread across disk drives in several ways:

Moving Databases *initlocation* allows you to create databases on different drives.

Moving Tables Symbolic links allow you to move tables and indexes to different drives. Movement should only be done while POSTGRESQL is shut down. Also, POSTGRESQL doesn't know about the symbolic links, so if you delete the table and recreate it, it will be created in the default location for that database. In 7.1, *pg_database.oid* and *pg_class.relfilenode* map database, table, and index names to their numeric file names.

Moving Indexes Symbolic links allow moving indexes to different drives from their heap tables. This allows an index scan to be performed on one disk while a second disk performs heap lookups.

Moving Joins Symbolic links allow the movement of joined tables to separate disks. If tables A and B are joined, lookups of table A can be performed on one drive while lookups of table B can be done on a second drive.

Moving Write-Ahead Log Symbolic links can be used to move the *pg_xlog* directory to a different disk drive. (Pg_xlog exists in POSTGRESQL releases 7.1 and later.) Unlike other writes, POSTGRESQL log writes must be flushed to disk before completing a transaction. The cache cannot be used to delay these writes. Having a separate disk for log writes allows the disk head to stay on the current log cylinder so writes can be performed without head movement delay. (You can use the *postgres -F* parameter to prevent log writes from being flushed to disk, but an operating system crash may require a restore from backup.)

Other options include the use of RAID features to spread a single filesystem across several drives.

Checkpoints

When write-ahead log files fill up, a checkpoint is performed to force all dirty buffers to disk so the log file can be recycled Checkpoints are also performed automatically at certain intervals, usually every 5 minutes. If there is a lot of database write activity, the write-ahead log segments can fill too quickly, causing excessive slowness as all dirty disk buffers are flushed to disk.

Determining Checkpoint Frequency

Checkpoints should happen every few minutes. If they happen several times a minute, performance will suffer. To determine checkpoint frequency, you must first enable log file timestamps in *postgresql.conf* with:

```
log_timestamp = true
```

After you force the postmaster to recognize this *postgresql.conf* change with *pg_ctl reload,* you will see these lines in the PostgreSQL server log file:

```
2002-02-11 21:17:32 DEBUG: recycled transaction log file 0000000000000000
2002-02-11 21:17:33 DEBUG: recycled transaction log file 0000000000000001
2002-02-11 21:17:33 DEBUG: recycled transaction log file 0000000000000002
2002-02-11 21:18:13 DEBUG: recycled transaction log file 0000000000000003
2002-02-11 21:18:13 DEBUG: recycled transaction log file 0000000000000004
2002-02-11 21:18:13 DEBUG: recycled transaction log file 0000000000000005
```

Measure the duration between checkpoints to determine if you are checkpointing too frequently. As you can see from the example above, checkpoints are happening ever fourty seconds, which is too frequent for good performance. Do not be alarmed if there are several log entries with identical timestamps, as shown above. Multiple messages are often generated by a single checkpoint.

Reducing Checkpoint Frequency

Reducing checkpoint frequency involves increasing the number of write-ahead log files created in *data/pg_xlog*. Each file is 16 megabytes, so this does affect disk usage. The default setup uses a minimum number of log files. To decrease checkpoint frequency, you need to increase this parameter:

```
checkpoint_segments = 3
```

The default value is three. Increase it until checkpoints happen only every few minutes. Another log message that may appear is:

```
DEBUG: XLogWrite: new log file created - consider increasing WAL_FILES
```

This message indicates that the *wal_files* parameter should be increased in *postgresql.conf.*

Conclusion

Fortunately, POSTGRESQL doesn't require much tuning. Most parameters are automatically adjusted to maintain optimum performance. Cache size and sort size are two parameters administrators can control to make better user of available memory. Disk access can also be spread across drives. Other parameters may be set in *share/postgresql.conf.sample.* You can copy this file to *data/postgresql.conf* to experiment with some of POSTGRESQL's even more exotic parameters.