⛳ All in One, Tips & Tricks for tuning & Maintaining PostgreSQL for DBAs⛳

Part D (Final part):

Memory Sizing for PostgreSQL

A PostgreSQL database utilizes memory in two primary ways, each governed by distinct parameters:

- **Caching data and indexes.** PostgreSQL maintains a cache of both raw data and indexes in memory. The size of this cache is determined by the shared_buffers parameter. Data is stored in "pages," which are essentially chunks of data. This applies to both data files and indexes. The presence or absence of a page in memory determines if it's cached. While there are ample resources to delve into the intricacies of cache eviction algorithms and page structures, the key thing to remember is that PostgreSQL uses an LRU (Least Recently Used) system for cache eviction.
- **Working memory for intermediate data structures.** When processing queries, PostgreSQL often needs to create temporary data structures to aid in delivering results. This could be as part of join operations, sorting, or even generating aggregates. The amount of memory that can be allocated for these structures is managed by the work_mem parameter. For instance, consider the various join algorithms employed by PostgreSQL: many require memory for operations like sorting or creating a hash map. Similarly, aggregate functions like GROUP BY and DISTINCT, or queries using ORDER BY (that don't use an index for sorting), may also create temporary data structures that occupy memory. If a join or hash needs more memory than your work_mem setting, it will spill to disk, which will be much slower.

These two parameters (shared_buffers and work_mem) are especially important. We look at them in detail later in this series, as they’re convenient in understanding memory-related performance issues in PostgreSQL:

- The cache maintained in shared_buffers is crucial for speeding up data access. A low cache hit ratio indicates that the database has to frequently read from disk rather than from the faster memory cache. Increasing the shared_buffers might help, but ensure it's balanced with the OS cache.
- If complex operations (like sorts or joins) frequently spill to disk because they exceed the configured work_mem, performance will suffer. Increasing work_mem may help, but you must be cautious here, too, as setting it too high could eventually consume all your memory, especially with many concurrent connections.

It's worth noting that there are two layers of caching: the PostgreSQL memory cache, which is directly managed by the database and is the primary cache level where data and indexes are held, and the operating system cache. This is an auxiliary cache level managed by the operating system. While PostgreSQL isn't directly aware of what's cached here, this layer significantly accelerates file-reading operations by holding onto files recently read from disk.

Generally, if your running database has more memory, it has to read from (and sometimes even write to) disk less, which is a good thing. But extra memory is expensive. How can you know that you *have to* increase the amount of RAM on your machine?

- **Queries have unpredictable performance**. If a query is slow when run once and then fast on the second go, it could be a sign that the data it needs cannot stay in the cache consistently. However, there are other potential causes for this type of speed pattern (such as high query planning time, which is cached), so it's worth directly observing this difference by comparing shared read vs. shared hit in your query plan.
- **You have many big indexes**. If all of your queries can be satisfied with index scans, and all of your indexes can fit in memory, you will likely get *good enough* performance even if the database has to go to disk to fetch the whole returned row. Many users are aware that more indexes on a table mean that inserts, updates, and deletes have to do more work, but it's also the case that in order to properly take advantage of additional indexes, you may need to increase the memory on your machine so they can stay in the cache.

- **Your sorts are spilling to disk**. As we've mentioned before, if you see an External Sort or External Merge node while debugging slow queries, this means that the memory required for the operation exceeds the available work_mem. As a result, PostgreSQL had to write its intermediate data structure to disk and then clean up at the end.
- **You see a lot of DataFileRead wait events in pg_stat_activity**. Anytime a query reads from disk, a wait event is triggered. Directly observing these in pg_stat_activity or comparing the number of events over time can give you a clue that you may need to fiddle with your memory. Significant wait events are also a sign that sequential scans are happening, so make sure to eliminate the possibility that you're simply missing an index (or have written the query to make the use of the index impossible) before spending more money.

Lastly, here are a few extra tips on memory monitoring.

What is the difference Reindex and Vacuum in PostgreSQL?

Reindex rebuilds indexes while Vacuum reclaims storage occupied by dead tuples.

Can Reindexing improve query performance in PostgreSQL?

Yes, it can improve query performance by optimizing the index and reducing the amount of data that needs to be examined especially when fragmentation rate of index is more that 40%.

How can I check if my PostgreSQL database needs Reindexing?
Regular monitoring of index bloat can help identify the need for reindexing.
Understanding reindexing in PostgreSQL and its implications is crucial for managing and optimizing a PostgreSQL database effectively. As we've seen, it can be beneficial for query performance and overall database health.

Some hints about Reindexing:

You are unsure whether to rebuild an index or not, ask yourself the following:

• If you observe the bloat over time, does it continuously increase? If yes, that is bad, and a REINDEX is probably necessary.
• If the bloat stays high, but does not increase, REINDEX once and observe
• Does that improve the performance of your application?
• Does the index get bloated again until it is as bloated as before?
"No" to the second question would suggest to leave the index alone.
If you decide to REINDEX, explore if more aggressive autovacuum settings on the affected table can keep the bloat down.

Different between ccnew and ccold Suffix when reindex an index on PostgreSQL?

Rebuilding Indexes Concurrently

Rebuilding an index can interfere with regular operation of a database.
Normally PostgreSQL locks the table whose index is rebuilt against writes and performs the entire index build with a single scan of the table. Other transactions can still read the table, but if they try to insert, update, or delete rows in the table they will block until the index rebuild is finished. This could have a severe effect if the system is a live production database. Very large tables can take many hours to be indexed, and even for smaller tables, an index rebuild can lock out writers for periods that are unacceptably long for a production system.

PostgreSQL supports rebuilding indexes with minimum locking of writes. This method is invoked by specifying

the CONCURRENTLY option of REINDEX.
When this option is used, PostgreSQL must perform two scans of the table for each index that needs to be rebuilt and wait for termination of all existing transactions that could potentially use the index. This method requires more total work than a standard index rebuild and takes significantly longer to complete as it needs to wait for unfinished transactions that might modify the index. However, since it allows normal operations to continue while the index is being rebuilt, this method is useful for rebuilding indexes in a production environment. Of course, the extra CPU, memory and I/O load imposed by the index rebuild may slow down other operations.

The following steps occur in a concurrent reindex.
Each step is run in a separate transaction. If there are multiple indexes to be rebuilt, then each step loops through all the indexes before moving to the next step.

1. A new transient index definition is added to the catalog pg_index. This definition will be used to replace the old index. A SHARE UPDATE EXCLUSIVE lock at session level is taken on the indexes being reindexed as well as their associated tables to prevent any schema modification while processing.

2. A first pass to build the index is done for each new index. Once the index is built, its flag pg_index.indisready is switched to “true” to make it ready for inserts, making it visible to other sessions once the transaction that performed the build is finished. This step is done in a separate transaction for each index.

3. Then a second pass is performed to add tuples that were added while the first pass was running. This step is also done in a separate transaction for each index.

4. All the constraints that refer to the index are changed to refer to the new index definition, and the names of the indexes are changed. At this point, pg_index.indisvalid is switched to “true” for the new index and to “false” for the old, and a cache invalidation is done causing all sessions that referenced the old index to be invalidated.

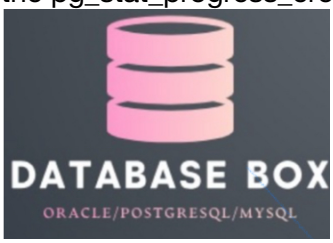
5. The old index have pg_index.indisready switched to “false” to prevent any new tuple insertions, after waiting for running queries that might reference the old index to complete.

6. The old indexes are dropped. The SHARE UPDATE EXCLUSIVE session locks for the indexes and the table are released.
If a problem arises while rebuilding the indexes, such as a uniqueness violation in a unique index, the REINDEX command will fail but leave behind an “invalid” new index in addition to the pre-existing one. This index will be ignored for querying purposes because it might be incomplete; however it will still consume update overhead. The psql \d command will report such an index as INVALID:

```
postgres=# \d tab
      Table "public.tab"
Column |  Type   | Modifiers
--------+---------+-----------
col    | integer |
Indexes:
   "idx" btree (col)
   "idx_ccnew" btree (col) INVALID
```

If the index marked INVALID is suffixed **ccnew**, then it corresponds to the transient index created during the concurrent operation, and the recommended recovery method is to drop it using DROP INDEX, then attempt REINDEX CONCURRENTLY again. If the invalid index is instead suffixed **ccold**, it corresponds to the original index which could not be dropped; the recommended recovery method is to just drop said index, since the rebuild proper has been successful.

Regular index builds permit other regular index builds on the same table to occur simultaneously, but only one concurrent index build can occur on a table at a time. In both cases, no other types of schema modification on the table are allowed meanwhile. Another difference is that a regular REINDEX TABLE or REINDEX INDEX command can be performed within a transaction block, but REINDEX CONCURRENTLY cannot.
Like any long-running transaction, REINDEX on a table can affect which tuples can be removed by concurrent VACUUM on any other table.
REINDEX SYSTEM does not support CONCURRENTLY since system catalogs cannot be reindexed concurrently.
Furthermore, indexes for exclusion constraints cannot be reindexed concurrently. If such an index is named directly in this command, an error is raised. If a table or database with exclusion constraint indexes is reindexed concurrently, those indexes will be skipped. (It is possible to reindex such indexes without the CONCURRENTLY option.)
Each backend running REINDEX will report its progress in the pg_stat_progress_create_index view.



Recovery Corrupted system indexes?

If you suspect corruption of an index on a user table, you can simply rebuild that index, or all indexes on the table, using REINDEX INDEX or REINDEX TABLE.
Things are more difficult if you need to recover from corruption of an index on a system table. In this case it's important for the system to not have used any of the suspect indexes itself. (Indeed, in this sort of scenario you might find that server processes are crashing immediately at start-up, due to reliance on the corrupted indexes.) To recover safely, the server must be started with the -P option, which prevents it from using indexes for system catalog lookups.
One way to do this is to shut down the server and start a single-user PostgreSQL server with the -P option included on its command line.
To start a single-user mode server, use a command like

postgres --single -D /usr/local/pgsql/data other-options my_database

Ubuntu has conf files spread over several places so:
/usr/lib/postgresql/13/bin/postgres --single -D /var/lib/postgresql/13/main -c "config_file=/etc/postgresql/13/main/postgresql.conf"

For more info see : https://www.postgresql.org/docs/current/app-postgres.html

Then, REINDEX DATABASE, REINDEX SYSTEM, REINDEX TABLE, or REINDEX INDEX can be issued, depending on how much you want to reconstruct.
If in doubt, use REINDEX SYSTEM to select reconstruction of all system indexes in the database. Then quit the single-user server session and restart the regular server. See the **postgres** reference page for more information about how to interact with the single-user server interface.
Alternatively, a regular server session can be started with -P included in its command line options. The method for doing this varies across clients, but in all libpq-based clients, it is possible to set the PGOPTIONS environment variable to -P before starting the client. Note that while this method does not require locking out other clients, it might still be wise to prevent other users from connecting to the damaged database until repairs have been completed.
REINDEX is similar to a drop and recreate of the index in that the index contents are rebuilt from scratch. However, the locking considerations are rather different.

REINDEX locks out writes but not reads of the index's parent table. It also takes an ACCESS EXCLUSIVE lock on the specific index being processed, which will block reads that attempt to use that index. In particular, the query planner tries to take an ACCESS SHARE lock on every index of the table, regardless of the query, and so REINDEX blocks virtually any queries except for

some prepared queries whose plan has been cached and which don't use this very index. In contrast, DROP INDEX momentarily takes an ACCESS EXCLUSIVE lock on the parent table, blocking both writes and reads. The subsequent CREATE INDEX locks out writes but not reads; since the index is not there, no read will attempt to use it, meaning that there will be no blocking but reads might be forced into expensive sequential scans.

A deep look and check the indexes and find the problem

Does Fragmentation is an important option to consider an index to reindex?

You can use pgstatindex from the pgstattuple extension to get the leaf fragmentation of all B-tree indexes:

```
 SELECT i.indexrelid::regclass,
     s.leaf_fragmentation
FROM pg_index AS i
  JOIN pg_class AS t ON i.indexrelid = t.oid
  JOIN pg_opclass AS opc ON i.indclass[0] = opc.oid
  JOIN pg_am ON opc.opcmethod = pg_am.oid
  CROSS JOIN LATERAL pgstatindex(i.indexrelid) AS s
WHERE t.relkind = 'i'
 AND pg_am.amname = 'btree';
```

This will scan through all these indexes and might take a long time.

To monitor index fragmentation in PostgreSQL, you can use the following SQL script to check the pgstattuple extension:

```
SELECT
   relname,
   indexrelname,
   idx_scan,
   pg_size_pretty(idx_tup_read * pg_relation_size(indexrelid)) AS "Index Size",
   pg_size_pretty(pg_relation_size(indexrelid)) AS "Table Size"
FROM pg_index
JOIN pg_class ON pg_index.indrelid = pg_class.oid
JOIN pg_class ci ON pg_index.indexrelid = ci.oid
LEFT JOIN pg_stat_all_indexes ON pg_stat_all_indexes.indexrelid = ci.oid
ORDER BY relname, indexrelname;
```

You normally don't need to reindex a PostgreSQL index. They will become somewhat fragmented over time, but that is normal.

That said, you can use the pgstatindex function from the pgstattuple extension to examine an index:

```
CREATE EXTENSION pgstattuple;

SELECT avg_leaf_density FROM pgstatindex('items_1_pkey');

avg_leaf_density
------------------
           88.92
(1 row)
```

That index is perfectly dense, the density is about the same as the fillfactor (90). Consider reindexing if the value drops below 20 or so.

Note that it is normal for an index to be up to 70% fragmented.

Find high costly indexes / Bad indexes:

SQL> SELECT schemaname, relname, seq_scan, seq_tup_read, idx_scan, seq_tup_read / seq_scan AS avg FROM pg_stat_user_tables WHERE seq_scan> 0 ORDER BY seq_tup_read DESC; schemaname | relname | seq_scan | seq_tup_read | idx_scan | avg
------------+-----------------+----------+--------------+----------+---------
Public | pgbench_accounts | 954 | 5050000000 | | 5293501
Public | pgbench_branches | 254 | 25400 | 0 | 100
public | pgbench_tellers | 1 | 1000 | 252 | 1000
(3 rows)

This one really works magic. It reveals the tables that have been most frequently hit by sequential scans and the average number of rows that a sequential scan has hit. In the case of our top query, a sequential scan has typically read 5 million rows, with no use of indexes. This shows us unequivocally that there is a problem with this table. A straightforward d will reveal the most glaring issues if you happen to be familiar with the application. In order to confirm our suspicion, let's look further:

pg_stat_statements: Finding slow queries

As already mentioned, pg_stat_statements are the gold standard for identifying slow queries. Typically, some of the worst pg_stat_statements queries have a high ranking for the tables that are present in pg_stat_user_tables.

The reality will be revealed by the response to this query:

SQL> SELECT query, total_exec_time, calls, and mean_exec_time FROM pg_stat_statements ORDER BY total_exec_time DESC;
---+-------------------------------------
query | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
total_exec_time | 433708.0210760001
calls | 252
mean exec_time | 1721.0635756984136
---+-------------------------------------
query | SELECT abalance FROM pgbench_accounts WHERE aid = $1
total_exec_time | 174819.2974120001
calls | 252
mean_exec_time | 693.7273706825395 …

Wow, the top query takes 1.721 seconds on average to execute! That is a lot.
When you look at the query, you can see that it only has a simple WHERE clause that filters on "aid."
When we look at the table, we see that there is no index on "aid," which is disastrous for performance.

The exact same issue will be found if the second query is investigated further.

Improve your PostgreSQL indexing and benchmarking

Let's deploy the index, reset pg_stat_statements, and PostgreSQL's standard system statistics:

```
SQL> CREATE UNIQUE INDEX idx_accounts ON pgbench_accounts (aid);
CREATE INDEX

SQL> SELECT pg_stat_statements_reset();
pg_stat_statements_reset
-------------------------
(1 row) SQL>
SELECT pg_stat_reset(); pg_stat_reset
---------------
(1 row)
```

After create index, query will be very fast.
The lesson to be learned from this is that even one missing PostgreSQL index in a crucial location can completely wreck a database and keep the entire system busy without producing any useful performance.

The approach we took to the issue is what is crucial to keep in mind. An excellent indicator to use when determining where to look for issues is pg_stat_user_tables. After that, you can search through pg_stat_statements for the worst queries. The key is to sort by total_exec_time DESC.

HUGE PAGES Concept & Configuration:

For people not familiar with the meaning of huge pages, here is a short overview of what they are and which problems they try to solve.
On any modern system, applications don't use physical memory directly. Instead, they use a virtual memory address model to make it easier to handle memory allocation and avoid the complexity of computing and mapping physical addresses into the application memory space. The virtual memory address model, however, requires the computer to translate virtual addresses into their corresponding physical memory addresses as soon as any application reads or writes to or from memory. This mapping is organized in the so-called page tables, a hierarchically-organized lookup table.

The TLB (Translation Lookaside Buffer) and Performance

Memory allocations on any common operating system are done by allocating single or multiple pages of available memory. These pages have a specific size, mostly depending on the operating system. On Linux, the default page size used by memory allocation are 4kB. So someone can imagine that allocating very large amounts of memory requires a high number of pages to be held in the page table, in order to map them to their corresponding physical addresses. Looking up virtual addresses requires multiple memory accesses, which are very costly when compared to CPU speed. Thus, modern CPUs maintain a so-called Translation Lookaside Buffer (TLB), which is basically a cache that speeds up translation between virtual and physical memory.
The size of a TLB is constrained to a few thousand entries, and especially on systems with high memory-specific memory workload, very soon becomes a bottleneck. If, for example, a database like PostgreSQL uses a shared buffer pool to implement Inter Process Communication (IPC), block layer access, and other functionality over it, this very quickly puts a high amount of pressure on the TLB, since the number of virtual address mappings is limited. Especially when using dozens of gigabytes for the shared buffer pool, such instances will suffer from performance hits because of this, depending on their workload.

The solution: Huge Pages

On modern operating systems like Linux this problem can be mitigated by using Huge Pages. Since page tables are organized hierarchically, they enable us to summarize allocations in much larger pages than the default. The size of huge pages are architecture-dependent, on x86 systems we can usually expect 2MB or 1GB sizes, IBM POWER allows 64kB, 16MBand 16 GB.
Lookups for specific virtual addresses of memory allocations are then much faster and hopefully more independent of the entries found in the TLB.
On x86, when configuring huge pages, the 2MB page size is the default. You can easily get your system current settings via /proc/meminfo:
[shell]
% cat /proc/meminfo | grep -i HugePage
AnonHugePages: 161792 kB
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
[/shell]
HugePages_Total says no huge pages are configured at the moment, accordingly HugePages_Rsvd,HugePages_Free and HugePages_Surp are equal to zero. Hugepagesizeshows the currently configured default size for huge pages in the kernel. When configuring huge pages later, we will revisit the value of these settings.

POSTGRESQL USAGE OF HUGE PAGES

PostgreSQL uses huge page allocations only for its shared buffer pool. Dynamic memory allocation for e.g. work_mem, maintenance_work_mem, autovacuum_work_mem doesn't use them. Also, dynamic shared memory segments used to transfer data and communication between parallel query workers don't request them automatically, with one exception, which I'll describe in the next paragraph about configuring PostgreSQL for huge pages on Linux. It's also necessary to have shared buffers allocated in so-called mmap'ed segments, which are controlled by the shared_memory_type parameter. This is the default setting on Linux. System V shared memory is only used by PostgreSQL to initialize a very small shared memory area; the buffer pool itself is allocated via mmap.

CONFIGURE YOUR SYSTEM

Huge Pages aren't configured on recent Linux Distributions out of the box. There are usually two ways to teach your Linux system to provide them to your applications:
- Transparent Huge Pages (THP)
- Explicitely configured pools of huge pages for applications
The first possibility via THP isn't recommended for PostgreSQL on Linux, since it doesn't allocate huge pages directly but provides them to applications as they need them. THP are swappable and considered a slower option compared to explicit configured huge pages. Other operating systems like FreeBSD provide better support in this case.

AN EASY EXAMPLE

So we are going to focus on explicit huge pages support for PostgreSQL. In the following sections, examples are all done on a Rocky Linux 9 system, but should apply to all other Linux systems accordingly. To configure explicit huge pages, we need to know the computed size of shared memory and additional shared allocations in advance. As of PostgreSQL 15, this is very easy (but unfortunately requires a PostgreSQL instance to be down). With the PostgreSQL 15 Community RPM installation from yum.postgresql.org it looks as follows:

[shell]
% su - postgres
% /usr/pgsql-15/bin/postgres --shared-buffers=20GB -D $PGDATA -C
shared_memory_size_in_huge_pages
10475
[/shell]
Again, the above code doesn't start PostgreSQL, but calculates the value of shared_memory_size_in_huge_pages and prints the result to the terminal. This also gives us some flexibility to calculate this value according to specific configuration settings we can provide to the postgresbinary as command line options. The example above specifies --shared-buffers as a command line option, overriding any other setting in the configuration files (like postgresql.conf).
If PostgreSQL cannot allocate the requested amount of shared buffers with huge pages, it falls back to normal page size allocations. This is controlled by the configuration parameter huge_pages, which can have the following values:

- try: Try to allocate huge pages, fall back to normal page allocation if not possible (the default setting currently)
- off: Never try to request huge page allocation
- on: Force shared buffer pool allocations with huge pages

If we have finally found our settings to start with, we need to reserve the required pool of huge pages by the kernel. As user root and using the default huge page size, this can be done during runtime by the following command:
[shell]
% echo 10475 > /proc/sys/vm/nr_hugepages
[/shell]
The memory settings then can be used to confirm what we have in mind:
[shell]
grep -i hugepage /proc/meminfo
AnonHugePages: 124928 kB
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 10475
HugePages_Free: 10475
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
[/shell]
Pools for specific huge pages must be reserved as a continuous memory area, so the kernel needs a memory area which is available for the reservation. If that's not possible, far less memory might be available to the pool. In this case, the HugePages_Totalnumber is lower than we requested. Experimenting with values that are too large can also drive a system into an out-of-memory situation, rendering it unavailable in the worst case. So I recommend carefully checking memory usage if the system hosts other applications or users.
Setting the number of pages like this, though, is not persistent across system reboots. To survive them, the easiest way is to put a configuration into /etc/sysctl.d/ with the right settings and reboot:
[shell]
% cat /etc/sysctl.d/01-nr-hugepages.conf
vm.nr_hugepages=10475
[/shell]
PostgreSQL is then configured to request huge pages. In this example, we adjust the postgresql.confconfiguration file to force PostgreSQL to perform allocations accordingly:
[text]
shared_buffers = 20GB
huge_pages = on
huge_page_size = 0 #use default kernel setting
[/text]
PostgreSQL must be started (or restarted) and should allocate its shared buffer pool via the default huge page size. /proc/meminfo changes its figures. Now it looks like it does here in the following output:

[shell]
grep -i hugepage /proc/meminfo
AnonHugePages: 176128 kB
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 10475
HugePages_Free: 10264
HugePages_Rsvd: 10264
HugePages_Surp: 0
Hugepagesize: 2048 kB
[/shell]
Note the HugePages_Rsvd figure: It reflects the number of pages reserved due to the allocations requested. You can use the pmap tool with the PID of the postmaster process to get the allocated memory mapped buffer pool segment:

pmap $(head -n1 /var/lib/pgsql/15/data/postmaster.pid) | grep huge
00007f11a5400000 21452800K rw-s- anon_hugepage (deleted)

This output shows the address where the memory mapped buffer pool area is mapped, its size and permissions. The (deleted) indicates this memory mapped allocation is already unlinked, which is a common trick to make it disappear as soon as the process owning it exits.

DYNAMICALLY ALLOCATED SHARED MEMORY AND HUGE PAGES

In the section on how PostgreSQL uses huge pages, we owe an explanation to the exception mentioned there. In general, dynamic shared memory allocations are done on purpose, for example when parallel query workers process a parallelized query. These dynamically allocated memory areas aren't done by mmap by default, instead PostgreSQL uses the posixAPI via shm_open() on Linux by default. In this case, allocations don't use huge pages. However, when having parallel queries, memory pressure can arise there as well, especially when large amount of data needs to be processed.
PostgreSQL starting with version 14 provides the configuration parameter min_dynamic_shared_memory. When configured, the specified value defines the additional space in the shared memory created at server startup that should be used for parallel query workers. Since it is allocated along with the buffer pool, it also uses huge pages for allocation. We can extend the example above and re-calculate the setting with 2GB of additional shared memory for use for parallel queries:
[shell]
/usr/pgsql-15/bin/postgres -D $PGDATA --min-dynamic-shared-memory=2GB --shared-buffers=20GB -C shared_memory_size_in_huge_pages
11499
[/shell]
Again, the PostgreSQL instance needs to be stopped to execute the command successfully. Compared to the previous example, the number for huge pages to be reserved has now increased by an additional 2GB.
Reference:
https://www.cybertec-postgresql.com/

WHAT ABOUT POSTGRESQL VERSIONS OLDER THAN 15?

These versions are a little bit more difficult to handle. Particularly calculations on how many huge pages need to be reserved are much trickier. The best solution so far is just trying to start PostgreSQL manually with huge_pages=on and some debugging parameters. In this case PostgreSQL prints the following error message if it is not able to allocate the requested shared memory size:

[shell]
FATAL: could not map anonymous shared memory: Cannot allocate memory
HINT: This error usually means that PostgreSQL's request for a shared memory segment exceeded available memory, swap space, or huge pages. To reduce the request size (currently 21965570048 bytes), reduce PostgreSQL's shared memory usage, perhaps by reducing shared_buffers or max_connections.
LOG: database system is shut down
[/shell]
The interesting thing is the *HINT* part of the message, telling us that PostgreSQL wants to allocate 21965570048 bytes of shared memory. This value can be used to calculate the number of pages ourselves. The problem with this number is that this is more than we requested when specifying shared buffers to 20GB. This is because PostgreSQL also needs additional shared memory for other internal processes, like *wal_buffers* or *max_locks_per_transactions*. Extension might require additional shared
memory, too. PostgreSQL internally uses a formula that corresponds approximately to the following
[text]
((shm size + (BLKSZ - (shm size % BLKSZ))) / (huge page size * 1024)) + 1 = num huge pages
[/text]
Standard PostgreSQL installations are always compiled with BLKSZ equal to 8192, specifying the requested huge page size gives then the following result:
[text]
((21965570048 + (8192 - (21965570048 % 8192))) / (2048 * 1024)) + 1 = 10475
[/text]

CONFIGURATION WITH GIGANTIC HUGE PAGES

Things get even more complicated if you want PostgreSQL to use a page size of 1GB. Since this is not the default on Linux, configuration requires more steps and more careful thought in such a case.

Alireza Kamrani
RDBMS Technical Consultant.