## 理解Postgres中的缓存--深度指南

翻译者：LittleWuCoding

**缓存**

众所周知，缓存是提高数据库性能的一个重要手段。本文着重讲一讲PostgreSQL中的缓存相关的东西。当然万变不离其宗，原理都是共同的，理解了这些，你也很容易把它运用到其它数据库中。

**缓存是什么以及我们为什么需要缓存**

不同的计算机组件运行的速度是不一样的，他们的差距很大，我们人类在理解计算机所能达到的尺度上的数字方面非常差。

从下表我们就可以有一个清晰的认识。这些数字是按人的感知估计的。

|  |  |  |
| --- | --- | --- |
| **Access type** | **Actual time** | **Approximated time** |
| 1 CPU cycle | 0.3 ns | 1 s |
| Level 1 cache access | 0.9 ns | 3 s |
| Level 2 cache access | 2.8 ns | 9 s |
| Level 3 cache access | 12.9 ns | 43 s |
| Main memory access | 120 ns | 6 min |
| Solid-state disk I/O | 50-150 μs | 2-6 days |
| Rotational disk I/O | 1-10 ms | 1-12 months |
| Internet: SF to NYC | 40 ms | 4 years |
| Internet: SF to UK | 81 ms | 8 years |
| Internet: SF to Australia | 183 ms | 19 years |
| OS virtualization reboot | 4 s | 423 years |
| SCSI command time-out | 30 s | 3000 years |
| Hardware virtualization reboot | 40 s | 4000 years |
| Physical system reboot | 5 m | 32 millenia |

在数据库系统内部数据传输时(这里我们不讨论网络延迟这种)，我们讨论数据传输瓶颈一般认为的就是磁盘I/O，因此我们这里也主要把关注点放在这里。

我们知道，大多数OLTP工作负载是随机的I/O，但是从磁盘获取非常缓慢。为了克服这个问题，和其它现有的数据库系统差不多，Postgres也把数据缓存到RAM(也就是我们说的内存)以提高性能。

因此，本文提到的缓存主要是软件意义上的缓存，不同于硬件意义上的CPU缓存，硬盘缓存这样的实体，主要是指软件系统在内存中开辟的用以提高数据复用等目的的一块内存区域，由软件自身通过特定算法实现调度。

**了解术语**

在我们继续说下去之前，我们需要一些基础知识。因此我推荐阅读这些：

关于PostgreSQL 的物理存储：

<http://rachbelaid.com/introduction-to-postgres-physical-storage>

以及inter db中的介绍：

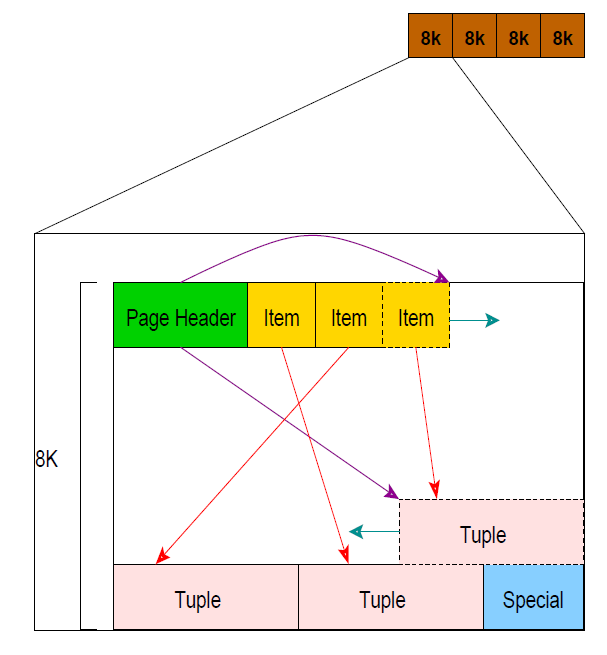
<http://www.interdb.jp/pg/pgsql01.html>

还有官方手册：

<http://www.postgres.cn/docs/9.5/storage.html>

以上文档我推荐你依次阅读。当然，上面的文档你不完全懂或者只知道一点也没关系，我会试着讲的简单一点。

或者，你试着记住下面这张图？它大致描述了Postgres的数据库存储的抽象层，也就是我们经常说的page(大小为8KB)。我们后面会用到的。



**什么被缓存了?**

问的很好。我们把哪些东西缓存到RAM中？

Postgres缓存了这些：

***Table data***

也就是表的内容。

***Indexes***

索引也是以page存储的，它们和table一样被缓存到RAM中，在下面我们还会提到它。

***Query execution plans***

我前面的一些文章提到过一条Query送到数据库中，会经过查询分析，查询重写，查询规划和查询执行这几个阶段才能获得最后的输出。其中查询规划阶段是针对所查询SQL选择出一条最优的查询计划出来。

是的，这里Postgres也把这个查询计划缓存了下来。这个查询计划一般是在优化或者analyze时会用到。当然，除非我么遇到那种特别复杂或者重复的查询，否则我们一般不用太关注它。

当然你也可以查询pg\_prepared\_statements视图来看看当前session缓存的查询计划(查询计划只在当前session有效，在当前session结束时被释放)。

所以你们知道，下面我们主要讨论table和index在内存中的缓存机制。

**存储区域**

Postgres服务器内部有很多配置参数来帮助你优化服务器的各个方面，因此理解他们的用法和用途就显得尤为重要了。

那么对于缓存，最重要的参数莫过于**shared\_buffers**了。你可以参考以下文档。

<http://www.postgres.cn/docs/9.5/runtime-config-resource.html#RUNTIME-CONFIG-RESOURCE-MEMORY>

在源码中，我们叫它NBuffers，所有Postgres的共享数据都存在这里。

shared\_buffers所代表的内存区域可以看成是一个以8KB的block为单位的数组，即最小的分配单位是8KB。没错，这正好是一个page的大小。每个page以page内部的元数据(Page Header)互相区分。

这样，当Postgres想要从disk获取(主要是table和index)数据(page)时，他会(根据page的元数据)先搜索shared\_buffers，确认该page是否在shared\_buffers中，如果存在，则直接命中，返回缓存的数据以避免I/O。

如果不存在，Postgres才会通过I/O访问disk获取数据(显然要比从shared\_buffers中获取慢得多)。

**LRU/时钟交换缓存算法**

提到缓存以及缓存数据的读入和逐出，其中最有代表性的恐怕就是LRU算法了。

我们Postgres的缓存也是类似，但是是采用的更高级的LRU时钟扫描策略，由于shared\_buffers专门用于处理OLTP工作负载，因此几乎所有的数据的移动都在内存中处理。

好的，我们细细的讲一讲。

**缓冲区的分配**

我们知道Postgres是基于进程工作的系统，即对于每一个服务器的connection，Postgres主进程都会向操作系统fork一个新的子进程(backend)去提供服务。同时，Postgres本身除了主进程之外也会起一些辅助的进程。

因此，对于每一个connection的数据请求，对应的后端进程(backend)都会首先向LRU cache中请求数据页page(这个数据请求不一定指的是SQL直接查询的表或者视图的page，比如index和系统表)，这个时候就发起了一次缓冲区的分配请求。

那么，这个时候我们就要抉择了。如果要请求的block就在cache中，那最好，我们"pinned"这个block，并且返回cache中的数据。所谓的"pinned"我暂时找不到一个好的词来代替他，它指的是增加这个block的"usage count"(这个时候你可以翻翻LRU算法了)。

当"usage count"为0时，我们就认为这个block没用了，在后面cache满的时候，它就该挪挪窝了。

那也就是说，只有当buffers/slots已满的情况下，才会引发缓存区的换出操作。

**缓存区的换出**

决定哪个page该从内存中换出并写回到disk中，这是一个经典的计算机科学的问题。

一个最简单的LRU算法在实际情况下基本上很难work起来。因为LRU是要把最近最少使用的page换出去，但是我们没有记录上次运行时的状态。

因此，作为一个折中和替代，我们追踪并记录每个page的"usage count"，在有需要时，将那些"usage count"为0的page换出并写回到disk中。后面也会提到，脏页面也会被写回disk。

不考虑细节问题的话，高速缓存算法本身几乎不需要调整就可以用在Postgres的缓存策略中，而且比人们通常所想的要智能得多。

**脏页和缓存的换出**

我们一直在说的是查询语句(SELECT),那么对于DML语句情况又有什么不一样呢？

很简单，他们将数据也写回到相同的page。也就是说，如果该数据所在的页面就在内存的cache中，那就把数据写到cache中的这个page里，如果不在内存，那么就把数据所在的page从disk读取到cache中，然后再写到cache的page里。

这里就是脏页面的由来：page里面的内容被修改了，但是却没有写回disk。

说到写回的概念，我们又要引出另外的概念了：[WAL](http://www.interdb.jp/pg/pgsql09.html) 和 [checkpoints](http://www.postgres.cn/docs/9.5/sql-checkpoint.html)。

WAL叫预写日志，它追踪并记录数据库系统中所发生的一切动作。中心概念是数据文件（存储着表和索引）的修改必须在这些动作被日志记录之后才被写入，即在描述这些改变的日志记录被刷到持久存储以后。如果我们遵循这种过程，我们不需要在每个事务提交时刷写数据页面到磁盘，因为我们知道在发生崩溃时可以使用日志来恢复数据库：任何还没有被应用到数据页面的改变可以根据其日志记录重做（这是前滚恢复，也被称为REDO）。

而Checkpointer进程呢？这个周期有系统参数可以配置。检查点是在事务序列中的点，这种点保证被更新的堆和索引数据文件的所有信息在该检查点之前已被写入。在检查点时刻，所有脏数据页被刷写到磁盘，并且一个特殊的检查点记录将被写入到日志文件（修改记录之前已经被刷写到WAL文件）。简而言之，它定期的将脏页面写回到disk。

有了这两个进程，就可以保证数据库很快地从崩溃中恢复过来，而不用手忙脚乱地回放崩溃之前的所有操作。

这是页面从内存中被换出的最常见的方式，在典型的情况下，LRU换出几乎不会发生。

**从explain analyze命令理解缓存**

现在让我们再回到数据库本身，从一个用户的角度去理解这个cache。这个时候，explain命令是我们的好帮手，它可以帮助我们了解很多有关cache的细节信息。

例如我们有一个如下的查询：

performance\_test=# explain (analyze,buffers) select \* from users order by userid limit 10;

Limit (cost=0.42..1.93 rows=10 width=219) (actual time=32.099..81.529 rows=10 loops=1)

Buffers: shared read=13

-> Index Scan using users\_userid\_idx on users (cost=0.42..150979.46 rows=1000000 width=219) (actual time=32.096..81.513 rows=10 loops=1)

Buffers: shared read=13

Planning time: 0.153 ms

Execution time: 81.575 ms

(6 rows)

这里的"Shared read"代表数据来自disk而并非cache。但是，如果我们再次运行这个查询，我们就会发现它变成了"shared hit"，也就是说，它已经在cache中了。

performance\_test=# explain (analyze,buffers) select \* from users order by userid limit 10;

Limit (cost=0.42..1.93 rows=10 width=219) (actual time=0.030..0.052 rows=10 loops=1)

Buffers: shared hit=13

-> Index Scan using users\_userid\_idx on users (cost=0.42..150979.46 rows=1000000 width=219) (actual time=0.028..0.044 rows=10 loops=1)

Buffers: shared hit=13

Planning time: 0.117 ms

Execution time: 0.085 ms

(6 rows)

通过这种方式可以非常方便地了解从查询角度了解有多少数据是被缓存的，而不是通过OS/Postgres的一些复杂的观测方法。

我们举一个有关顺序扫描的例子：当前查询的表上没有索引，而且postgres必须从磁盘提取所有数据。

由于单个seq扫描可以擦除缓存中的所有数据，因此处理方式和我们预想的不太一样。它并不使用LRU/时钟扫描算法，而是直接分配了总共256KB(32\*8KB)的缓冲区。

下面的查询计划显示了它是如何处理的。

performance\_test=# explain (analyze,buffers) select count(\*) from users;

Aggregate (cost=48214.95..48214.96 rows=1 width=0) (actual time=3874.445..3874.445 rows=1 loops=1)

Buffers: shared read=35715

-> Seq Scan on users (cost=0.00..45714.96 rows=999996 width=0) (actual time=6.024..3526.606 rows=1000000 loops=1)

Buffers: shared read=35715

Planning time: 0.114 ms

Execution time: 3874.509 ms

再次执行该查询：

performance\_test=# explain (analyze,buffers) select count(\*) from users;

Aggregate (cost=48214.95..48214.96 rows=1 width=0) (actual time=426.385..426.385 rows=1 loops=1)

Buffers: shared hit=32 read=35683

-> Seq Scan on users (cost=0.00..45714.96 rows=999996 width=0) (actual time=0.036..285.363 rows=1000000 loops=1)

Buffers: shared hit=32 read=35683

Planning time: 0.048 ms

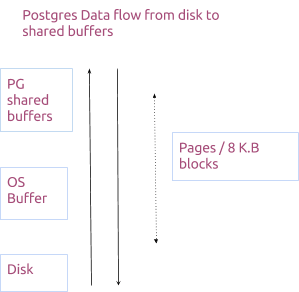
Execution time: 426.431 ms

我们可以看到总共32blocks(32 \* 8 = 256 KB)被移动到了内存中。理由？ 我们可以看看这里： src/backend/storage/buffer/README

**内存流和OS缓存**

Postgres作为一个跨平台的数据库，它的缓存机制在很大程度上依赖于它所安装的操作系统。

shared\_buffers实际上是复制了操作系统的功能。下面给出了数据如何流经postgres的典型图片。



这是令人困惑的，因为缓存居然是由操作系统和postgres共同管理的，我们知道这是有原因的。但是要谈操作系统缓存的话篇幅就很大了，需要另开一篇文章来讲了，但网上有很多资源可以利用。

但是请记住，操作系统缓存数据的原因与我们上面看到的相同，即为什么我们需要缓存？

我们可以将I/O分为两类，即读和写。 更简单的说，一个是数据从磁盘流向内存，另一个是数据从内存流向磁盘。

**读操作**

对于读来说，根据上面的流程图，数据从磁盘流向OS缓存，然后流向shared\_buffers。 我们已经讨论过这些页面如何被"pinned"在shared\_buffers上，直到它们被弄脏/解除固定。

有时，OS缓存和shared\_buffers可以保存相同的页面。你马上意识到这可能会导致空间浪费，但请记住操作系统缓存使用简单的LRU，而不是数据库优化的时钟扫描。 一旦这些页面在shared\_buffers上命中，就不会再去读取OS缓存，并且如果有任何重复，它们也能被轻松移除。

而且实际上，这两个内存区域上重复的页面并不多。因此，我们不必担心空间的浪费。相反，我们建议多花点时间仔细调整shared\_buffers。 因为无论shared\_buffers过大还是过小都会影响数据库的性能。。

我们将在下面讨论更多的优化方法。

**写操作**

而对于写，数据从内存流写入磁盘。 这是脏页面的概念的由来。

一旦页面被标记为脏，它将被刷新到操作系统缓存，然后写入磁盘。 这个时候操作系统有更多的自由来根据写入流量安排I/O。

如上所述，如果OS缓存大小过小，则不能重新排序写入和优化I/O。 这对写入较多的任务的影响尤其重要。 所以OS缓存的大小也很重要。

**初始化配置**

与其它许多数据库系统一样，在参数的配置上没有"银弹"。 因此，PostgreSQL提供了一个广泛兼容性而不是性能调优的基本配置。

数据库管理员有责任根据应用和工作负载调整配置。 但是，好在postgres的有一个很好的帮助文档。

一旦设置好了配置，我们就可以做负载/性能测试，来看看它是如何运行。因此最好始终进行实验来获得更适合的配置。

**随时随地优化**

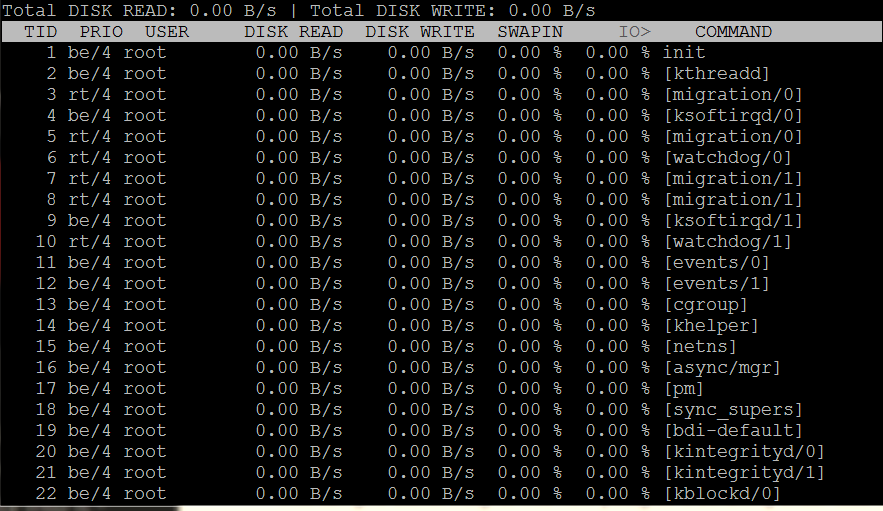
如果你不能量化一些东西的，那你不能很好地优化它。

对于Postgres，我们可以从两个方向来观察。

**操作系统层面**

对于postgres最适合哪个平台并没有普遍的一致意见，我假定您正在使用Linux系列的操作系统。把数据库放在Linux类系统上应该是很常见的做法。

在Linux系统上有一个名为iotop的工具，可以查看磁盘I/O。 与top类似，你只需运行命令iotop来观察写入/读取。



它可以提供有用的数据，了解postgres在负载下的行为方式，即根据产生的负载，有多少数据来自内存和磁盘。

**直接从postgres监控**

我们觉得直接从postgres监视某些东西总是比较好，而不是间接通过操作系统来观察。除非我们认为postgres本身有问题时，我们才会进行操作系统级别的监控，但这种情况很少。

在postgres内部，我们可以使用好几种工具来测量内存的性能。

**内置的EXPLAIN命令**

首先是默认是SQL EXPLAIN

详情见：<http://www.postgres.cn/docs/9.5/using-explain.html。>

它可以比任何其他数据库系统获得更多的信息，虽然有点难于理解，但是看看手册学习下你就能大致看懂。 另外不要错过这几个有用的标志(COSTS,BUFFERS,TIMING这些)，特别是我们之前看到的缓冲区。

**查询日志**

查询日志是了解系统内部情况的另一种方法。

我们可以设置参数[**log\_min\_duration\_statement**]参数来记录查询时间超过指定阈值的查询(慢查询)，而不是记录所有内容。

具体可以查看以下链接：

<http://www.postgres.cn/docs/9.5/runtime-config-logging.html#RUNTIME-CONFIG-LOGGING-WHEN>

**auto\_explain插件**

auto\_explain模块提供了一种方式来自动记录慢速语句的执行计划，而不需要手工运行EXPLAIN。这在大型应用中追踪未被优化的查询时有用。

具体可以查看以下链接：

<http://www.postgres.cn/docs/9.5/auto-explain.html>

**pg\_stat\_statements插件**

pg\_stat\_statements模块提供一种方法追踪一个服务器所执行的所有 SQL 语句的执行统计信息。

该模块必须通过在postgresql.conf的shared\_preload\_libraries 中增加pg\_stat\_statements来载入，因为它需要额外的共享内存。 这意味着增加或移除该模块需要一次服务器重启。

具体可以查看以下链接

<http://www.postgres.cn/docs/9.5/pgstatstatements.html>

这种方法的缺点是会有一定的性能损失，所以在生产系统中通常不推荐使用。

**pg\_buffer\_cache插件**

pg\_buffercache模块提供了一种方法实时检查共享缓冲区。

该模块提供了一个C函数pg\_buffercache\_pages，它返回一个记录的集合， 外加一个包装了该函数以便于使用的视图pg\_buffercache。

具体可以查看以下链接

<http://www.postgres.cn/docs/9.5/pgbuffercache.html>

**pg\_prewarm插件**

pg\_prewarm模块提供一种方便的方法把关系数据载入到操作系统缓冲区缓存或者 PostgreSQL缓冲区缓存。如果你认为"memory warm up"存在问题，那么这对调试非常有用。

具体可以查看以下链接

<http://www.postgres.cn/docs/9.5/pgprewarm.html>

还有一些，但我列出了最流行和最容易使用的用来了解Postgres缓存的工具。一般情况下，有了这些工具，就再也没有理由因为内存问题而导致数据库运行缓慢了。

原文链接：

<https://madusudanan.com/blog/understanding-postgres-caching-in-depth/>