知乎

已赞同 146

分享

记一次深入内核的数据库高并发性能优化实践

DolphinDB
已认证账号

关注

▲ 你关注的 南昌之星售票系统 赞同

前不久，我们接到客户长江电力的反馈，称在生产环境中进行高并发查询，例如包含数百个测点的**近千个并发作业**，在从近三月的数据中取数或聚合计算时，会CPU利用率却很低。

接到反馈后，我们的技术团队第一时间组织人员复现场景，本以为是一次普通平常的性能优化问题，没想到解决问题的过程堪比福尔摩斯探案。从脚本分析至核系统内核，我们抽丝剥茧，拨开层层迷雾，最终为客户揭开了性能不佳现象背后的谜团。

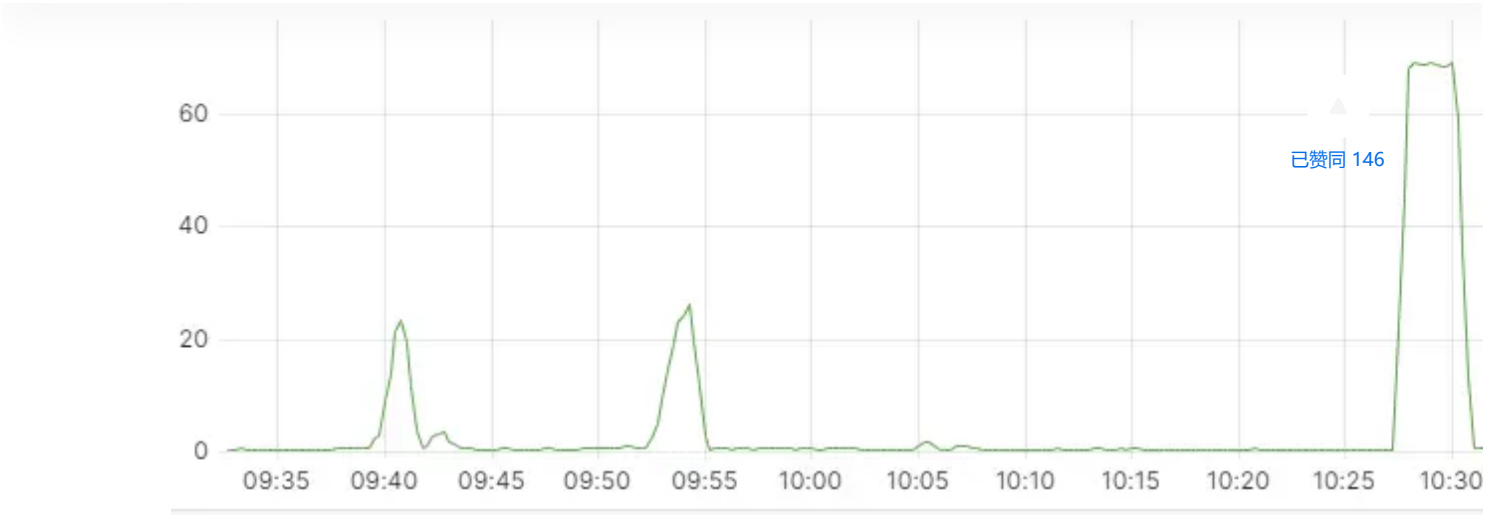
高并发查询性能不佳?

长江电力是中国最大的电力上市公司和全球最大的水电上市公司，其水力发电业务下属乌东德、白鹤滩、溪洛渡、向家坝、三峡、葛洲坝等六座水电站。作为长层数据库架构，近两年来，DolphinDB 一直为长江电力的水力发电项目提供**高性能的数据存储和计算的能力**支撑。

长江电力每个水电站包含多台机组，**全天候采集的数据测点峰值高达200万**。采集数据经 Kafka 实时推送至 DolphinDB 高可用集群。**目前每天产生的数据行数**

一般来讲，处理高并发查询下性能不足的问题，我们首先检查资源使用率，主要是 CPU 利用率、网络带宽、磁盘 I/O 三个方面。借助 dstat 工具，我们发现暴露出来的明显问题：上千个并发作业的情况下，集群的 CPU 利用率只有30%左右，网络和磁盘的资源也仍有冗余。那么如何提高 CPU 利用率，或者能够排对解决这个问题的预期。

知乎



CPU 利用率偏低

脚本分析无功而返

在明确以提升 CPU 利用率为目标的思路下，我们首先从查询的脚本入手分析问题。

第一个猜想：是否是查询并发度不够，或者是数据倾斜导致了某些节点闲置，从而导致了资源利用率低的情况。

我们首先分析了工作负载：在复现场景下，用户同时提交上千个查询任务，每个查询会涉及所有分区。由于 DolphinDB 会把总任务改写成针对每个分区的任务能够用到每个节点上所有的100个 worker，通过 DolphinDB Web 端监控确认，查询过程中每个节点的 worker 一直是满负载，等待队列上也一直有足够的任**作负载本身并发度不够导致资源利用率低的情况。**

第二步我们在脚本结构层面进行了优化分析。该场景下，查询语句的 where 条件包含排序键，在解析的过程中执行效率可能受到分区剪枝和谓词下推两个特性区表时，优化器可以通过分区剪枝消除不必要的分区，或者在执行查询时可以将过滤条件下推，直接让存储进程将符合范围的数据过滤掉，理论上这样可以让查

带着这个目标，我们对脚本进行了优化，但结果发现相同效果的查询语句，用到谓词下推和不能用到谓词下推的版本执行时间差别不大，并且没有经过优化的查是优化过脚本的两倍，这可能是因为没有谓词下推的脚本花在解压缩等工作上的时间更多，虽然涉及的 I/O 也更多，但总体还是表现为 CPU 利用率更高。**综上本优化不足导致的资源利用率低的情况。**

一波三折的火焰图：DolphinDB 代码分析

查询层面的分析似乎没有什么进展。我们打算换个思路，从代码层面入手，看看能否提高 CPU 利用率。

前面说到，由于进程中的 worker 没有闲置，但 CPU 利用率仍然不高。这时我们一般使用 **Off-CPU Flame Graph（火焰图）从锁争用和 I/O 两个方面来排查** Intel® VTune™ 的 Threading Analysisc 分析类型来生成火焰图。

VTune 用户态采集模式

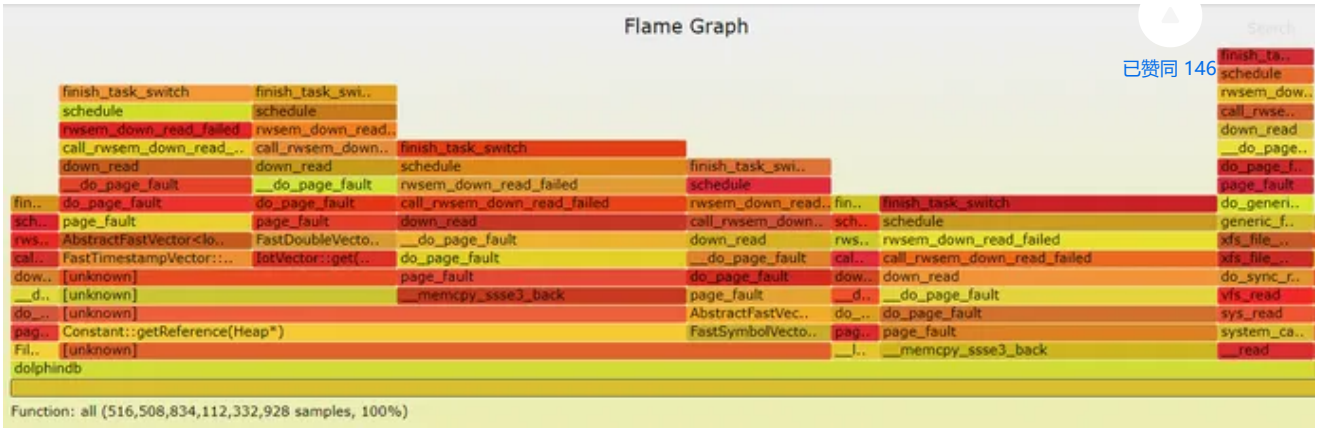
我们首先用VTune用户态采样模式进行分析。这个模式可以收集用户态线程在同步以及线程等待时的信息，给出每个线程详细的执行状态。**第一个发现**是 Cent争用问题频发，我们通过设置 `TCMALLOC_MAX_TOTAL_THREAD_CACHE_BYTES` 环境变量规避了内存分配器上锁争用的问题，但 CPU 利用率仍然没有变化。并且根据问题**转移到了DolphinDB TSDB 引擎层面。**

跟随采集结果的指示，**我们在 TSDB 引擎层面，先后对三处不同的锁进行了优化，但最终 CPU 利用率仍然没有变化。**此时，我们已经对 VTune 给出结果的准一开始使用 VTune 时，我们就发现了一个现象：在单独测试时，CPU 利用率一般是在20% - 30%，用了 VTune 之后，利用率直接降到了5%以内。这已经可以采集模式下自身开销极大，导致测试结果非常不准确。**因此，我们转而使用基于硬件事件的 VTune 采集方式，这种模式开销更小。**

VTune基于硬件事件的采集模式

知乎

有了之前用户态模式踩坑的经历，我们决定用其他工具再测试一次，做双重验证。于是我们又用 bcc offcputime 工具测试了一次，得到了与 VTune 相同的采



bcc offcpu 火焰图

结合 VTune 和 bcc offcpu 的测试结果，我们基本可以确定是 mmap_sem 锁竞争问题导致的 CPU 利用率低。

具体表现为：系统调用 mmap 和 munmap 时需要获取 mmap_sem 互斥锁，缺页异常处理时需要获取 mmap_sem 共享锁，两个地方会出现锁争用的问题，

深入内核，确认写锁来源

此时，我们对排查的方向有了一些动摇。按理来说，Linux 在缺页这种非常正常的操作上不应该存在这么严重的扩展性问题，虽然问题很少见，但是本着已经投到底的态度，我们决定继续深入下去。

因为上述测试本身确实会使用到大量内存，所以我们大胆排除了是因为内存分配器没有缓存，而导致频繁缺页，从而频繁读锁的可能性。

既然缺页获取 mmap_sem 读锁的情况无法避免，那么我们只能试着找到读锁来源，看看能否优化了。

然而我们在 VTune 的栈结果里并没有搜到 mmap_sem 获取写锁的栈，因此我们想到修改 bcc offcputime 脚本，把 finish_task_switch 事件改成在 down_w于测试的服务器内核版本太低（CentOS 7），只能显示内核栈，无法显示用户栈，这一操作无法实现。

直接查看写锁调用栈的方法暂时卡住了，在寻找新的思路同时，我们也搜索了大量 mmap_sem 相关的问题，寻找灵感。在阅读了大量材料后，确实发现有一些

综合这些讨论，我们猜测可能是内核里 mmap 操作太频繁，导致缺页时 mmap_sem 锁争用。

我们使用 strace 和 sysdig 这两个工具来确认 mmap 的调用频率。经过测试后，除了大致确认 mmap 调用确实过于频繁外，我们还从 sysdig 的输出中发现，着多次的 open 和 fstat操作（而且这些文件就是level file，不过mmap本身是匿名映射）。

这个现象让我们不禁怀疑，也许是因为 tcmalloc 里 mmap 调用太频繁而导致了资源争用问题。我们试图通过在 tcmalloc 中禁用 mmap（即使这样性能也许会题，但是即便如此，VTune 仍然显示瓶颈是缺页时 mmap_sem 锁争用。

经过一层层的排查和验证，我们在 VTune 的栈结果里发现了一处不常见的 mmap 相关调用栈，他将线索指向了 fseek，具体表现为在 TSDB 引擎使用 Dolp offset 读取数据调用 fseek 时，调用了 mmap，并且在文件流对象析构时调用了 munmap。这让我们做出了一个假设：是否是 Linux 上的文件操作在内部了冲突呢?

事实证明，正是这个差点被忽略的线索，让我们找到了解决问题的关键。

验证猜测：Linux文件操作对mmap的影响

我们首先查看了 fopen 的文档，得知如果通过 'm' 模式来打开文件，文件操作确实会调用 mmap，但是我们并没有使用该模式。然而 VTune 给出的栈信息明用了 mmap，为了进一步验证猜想，我们写了一个简单的程序，通过 gdb 在 mmap 设置断点的方式测试，发现了 fseek 内部确实会调用 mmap。

```
    printf("GNU libc version: %s\n", gnu_get_libc_version());

    FILE* fd = fopen("./result.csv", "rb");
    fseek(fd, 8192, SEEK_CUR);
    fclose(fd);
}
```



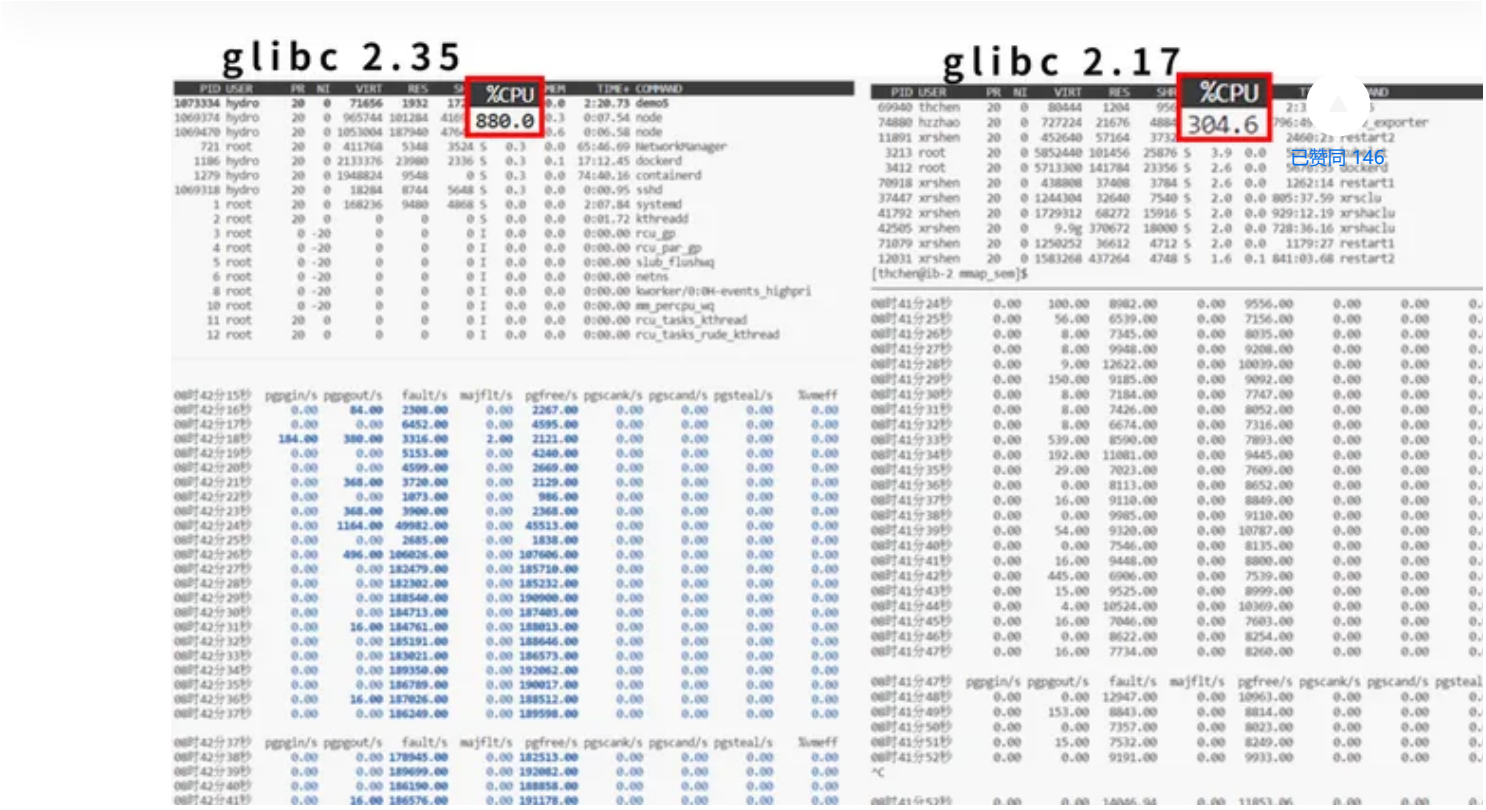
```
(gdb) bt
#0  0x00007ffff6ceef90 in mmap64 () from /lib64/libc.so.6
#1  0x00007ffff6c64021 in __GI__IO_file_doallocate () from /lib64/libc.so.6
#2  0x00007ffff6c72e57 in __GI__IO_doallocbuf () from /lib64/libc.so.6
#3  0x00007ffff6c6fc03 in __GI__IO_file_seekoff () from /lib64/libc.so.6
#4  0x00007ffff6c6d607 in fseek () from /lib64/libc.so.6
#5  0x00000000004006ec in main (argc=1, argv=0x7fffffffe228) at main.cpp:17
```

值得一提的是，我们非常幸运是直接在服务器上(CentOS 7, glibc 2.17)写的这个 demo，才发现了 fseek 的问题。如果是在本地机器上测试（Ubuntu 22, glib 错过了问题的关键。

通过 demo 测试，我们让一个线程一直执行 fopen/fseek/fclose，另外多个线程执行 mmap/memcpy/munmap（该测试为8个线程），结果发现在 glib cpu 利用率只有300%，而在 glibc 2.35中，cpu 利用率可以达到880%!

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>
#include <errno.h>
#include <vector>
#include <thread>
#include <stdio.h>
#include <stdlib.h>
#include <gnu/libc-version.h>

int main(int argc, char const* argv[])
{
    printf("GNU libc version: %s\n", gnu_get_libc_version());

    int tn = argc >= 2 ? std::atoi(argv[1]) : 1;
    std::vector<std::thread> ts;
    for (int i = 0; i < tn; i++) {
        ts.push_back(std::thread([](){
            while (true) {
                char* buf = (char*)mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
                for (int i = 0; i < 4096; i++) {
                    buf[i] = 1;
                }
                munmap(buf, 4096);
            }
        }));
    }

    while (true) {
        FILE* fd = fopen("./result.csv", "rb");
        fseek(fd, 8192, SEEK_CUR);
        fclose(fd);
    }

    for (auto& t : ts) {
        t.join();
```

知乎



glibc升级前后，CPU 利用率对比

真相大白！glbic 2.17 实现问题

到这里，我们基本确定了 **fseek 的拓展性问题。**

于是我们查看了 glibc 2.17和 glibc 2.35（此时已经发现了 glibc 2.35 fseek 不会调用 mmap 了） `__GI__IO_file_doallocate` 和 `_IO_setb` 的实现。 `__GI__I` glibc 2.17 会主动调用 mmap，而在 glibc 2.35 是通过 malloc 分配内存。类似的， `_IO_setb` 在 glibc 2.17 是主动调用 munmap，在glibc 2.35 是调用 free

我们立即修改了原测试环境的 glibc 版本进行原并发查询场景的测试，测试结果让我们心里的石头都落了地。

	查询时间范围	升级前查询第1次（ms）	升级前查询第2次（ms）	升级前查询第3次（ms）	升级后查询第1次（ms）	升级后查询第2次（ms）	升级后查询第3次（ms）	速度提
单查询	15天	19,295	17,319	17,629	7,491	7,964	8,256	2.
	30天	27,387	25,027	29,249	13,666	15,851	16,279	1.
	3月	73,203	153,747	63,653	72,249	55,105	44,839	1.
20并发	15天	123,593			50,745			2.
	30天	218,241			68,567			3.
	3月	683,128			279,223			2.

至此，经过一步步的猜测、推理、验证和推翻后再验证， “真相”终于水落石出：在用户的高并发查询场景下，系统读取 levelfile 非常频繁，加上一些/proc文 了 fseek，glibc 2.17的实现问题更导致了这一操作会频繁调用 mmap 和 munmap ，进而获取 mmap_sem的写锁，使得 page fault 无法获取 mmap_sem 的 栈锁争用过高，最终产生了CPU利用率不高的问题，影响了高并发查询场景下的性能。

如果您也遇到这样的问题，快升级你的glibc 2.17吧！详细升级手册请见：

DolphinDB：基于 Glibc 版本升级的 DolphinDB 数据查询性能优化实践
15 赞同 · 4 评论 文章

结语：千淘万漉虽辛苦，吹尽狂沙始到金

知乎

无数次猜测、讨论、验证之后，留下的不只是一份解决方案，更是一份对技术极度热爱、对挑战毫不畏惧的承诺。现在如此，未来亦然，我们将保持对技术的[illegible]去。

同样，随着越来越多客户将 DolphinDB 部署在关键的生产系统上，我们面临的技术场景也越发丰富而充满挑战。如果你也有志于精进技术、[illegible]占研，欢迎你术团队，与我们一同探索技术的无限可能！

已赞同 146

[注1] mmap_sem 相关讨论

- Db2 LUW (Linux): poor IO performance with VERITAS File System (VxFS) if nommapcio mount option is not enabled (ibm.com)
- Re: [v8-dev] mmap contention (mail-archive.com)
- On the surprising behaviour of memory operations at high thread counts | by Fabien Reumont-Locke | Medium

编辑于 2023-11-28 11:20 · IP 属地浙江

高并发 数据查询 GLIBC

推荐阅读

数据库并发控制总结

1.数据库并发控制的作用1.1 事务的概念在介绍并发控制前，首先需要了解事务。数据库提供了增删改查等几种基础操作，用户可以灵活地

知乎

已赞同 146