# Polaris：带优先级的乐观并发控制方法

**摘要**

本复现工作来自于 SIGMOD2023 的一篇工作，数据库被广泛应用的今天，在不同的领域中都有对于本领域内不同事务的优先级定义，优先级机制决定了事务的执行顺序，高优先级的事务应该比低优先级的事务拥有更加多的提交机会。而在目前的有优先级机制的并发控制机制通常是通过利用锁机制来支持的，但是优先级机制在乐观并发控制中却很少有出现。Polaris [1] 通过在乐观并发控制机制中引入轻量级的锁，使得其吞吐量能够于乐观并发控制机制相差不多的情况下，能够支持优先级机制，保留了乐观并发控制机制在吞吐量中的优势。

**关键词：**优先级；乐观并发控制；事务处理

## 1 引言

在联机事务处理数据库系统中，支持事务的多个优先级是至关重要的。利用优先级机制能够使得某类事务优先于其他事务，例如：用户发起的事务应该优于后台系统事务；有等待时间限制的事务应该优于其他事务；用户可以通过分配优先级，令某些事务优先执行，提高了数据库系统使用的灵活性。优先级对于较少尾部延迟也十分有用，在面对一个多次被中断的事务时，优先级机制能够提高该事务的优先级，以此避免饥饿的发生。目前已经出现了支持优先级的数据库管理系统了。结合现实，在日常使用的过程中，只读事务通常在总的待执行事务中占比较大，因此使用乐观并发控制协议对事务进行处理能够极大的提高数据库的吞吐量。但是，在如今的乐观并发控制协议中，几乎没有能够支持的优先级机制的协议，因此很难用于许多实际应用场景。Polaris 引入了轻量级的锁来执行优先级，在面对同一优先级的事务，该协议与乐观并发控制协议并无不同，而当高优先级的事务进行需要进行写入时，低优先级事务不能进行写入，而在写入过程中，该事务有可能会被更高优先级的事务所抢占。在这种设计下，发现高优先级事务的 p999 尾部延迟要比低优先级事务低 13 倍。这种设计能够保证高优先级事务能够优先执行，同时能够保留乐观并发控制协议的大吞吐量的特性。

## 2 相关工作

数据库系统用户可能需要优先处理某些关键事务，通过给这些事务设定高优先级，优先把系统资源分配给这些用户，同时还需要关注饥饿问题，对于一些多次被中断的事务也应该给予高优先级。

在事务优先级机制的实现中，并发控制协议能够提到重要作用，并发控制协议通常分为乐观并发控制与悲观并发控制，其区别在于是否具有锁机制。在悲观并发控制机制中，两阶段锁机制是较为常用的（Two-Phase Locking）。

### 2.1 两阶段锁 (2PL)

锁机制被广泛的应用在并发控制中，利用锁机制，能够防止冲突事务同时对相同数据进行修改，避免读写冲突和写写冲突发生。两阶段锁是一种使用锁机制的并发控制方法，它能够把事务序列化。在两阶段锁中，事务执行分为增长阶段和收缩阶段，事务只能在增长阶段获取新锁，在收缩阶段释放锁。两阶段锁的方法可根据其处理死锁的方式进一步分类。

就死锁检测方案而言，当发现死锁时，会选择一个受害者事务并中止，以打破死锁。如果用户指定了事务的优先级，则会选择优先级最低的事务作为受害者。No-Wait、Wait-Die 和 Wound-Wait 是三种常见的死锁预防方案。在 2PL 的 No-Wait 变体中，当一个事务遇到冲突（即锁获取被拒绝）时，它无需等待就会自行终止。在 Wait-Die 方案下，每个事务在执行前都会收到一个唯一的时间戳，时间戳越小表示优先级越高。如果一个事务请求加锁但被拒绝，如果它的优先级高于锁持有者，它就会把自己放入锁等待队列。否则，它必须中止，以避免死锁。这是一种非抢占式协议。最后，在 Wound-Wait 方案中，每个事务都有一个基于时间戳的优先级。在发生冲突时，如果请求事务的优先级更高，它就会抢占锁，并中止锁持有者。否则，它就会进入锁的等待队列。

### 2.2 乐观并发控制协议 (OCC)

乐观并发控制 (Optimistic Concurrency Control,OCC) 是一种可以确保事务能够可序列化的事务并发控制处理协议。在 OCC 应用环境中，假设争用的现象很少发生，因此能够让冲突检测延后至事务执行结束后进行。在 OCC 中，事务会把自己所需要的数据拷贝到本地的副本中，事务执行过程中在本地的副本中进行修改，在执行结束后，事务进入临界区，判断是否可以序列化，若通过验证便能够提交，生成日志并把修改写入磁盘中。

OCC 的优点在于其避免了锁的开销，并且不需要复杂的算法进行死锁预防和死锁检测。但是其全局临界部分影响了整体的可拓展性。在此基础上，Silo [2] 数据对 OCC 进行了改进，在 OCC 的基础上为每条记录新增了一个 64 位的 TID，该 TID 包含了数据的版本号以及一个锁存位。再事务执行过程中，事务会保留其本地副本中的所有更新，并维护一个读集和一个写集，以跟踪其访问的记录。在验证阶段中首先获取验证写集中的记录是否存在冲突，写集的验证使用锁存位，若某记录的锁存位为 1 则代表该记录正在被其他事务修改，因此验证失败，然后进行读集的验证，验证读集只需要验证读集中的记录的版本号是否与磁盘中对应记录的版本号相同，若相同则通过验证。然后把数据更新到磁盘中，在更新过程中，为写集中的记录上锁，在更新结束后，释放锁存器。

### 2.3 饥饿处理以及长尾延迟

有些工作负载对延迟非常敏感。例如，大型在线服务会向工作服务器发出请求并收集响应。服务器必须在一定时限内做出响应，以保证服务的响应性 [7]。为防止无限期增长的延迟，并发控制协议应确保事务的执行不会出现饥饿。Wound-Wait 和 Wait-Die 就不会出现饥饿。

Wound-Wait 方案保持一个不变式，即当发生锁冲突时，最老的事务总是赢得锁。如果较早的事务请求锁，当前锁持有者将被中止。在最坏的情况下，一个反复被中止的事务会成为系统中最老的事务，并最终获得它想要的所有锁。在 Wait-Die 方案中，只有比锁持有者老事务才能等待锁，这就保证了锁队列不会随着更多新事务的到来而无限增长。相比之下，OCC 没有对饥饿的处理。OCC 将冲突检测推迟到提交阶段，因此无法保证事务能通过验证并提交。如果读取集中的记录不断被其他事务修改，当前事务可能会无限期地中止。

图 1显示了 Silo 和三种不同的 2PL 变体在高竞争环境下的尾延迟和吞吐量结果。虽然 Silo 实现了更高的吞吐量，但其 p999 延迟却高于 Wound-Wait 和 Wait-Die。如果比较 p9999 延迟，差距会更大。

通过在 OCC 的基础上实现对优先级的支持，能够使一些关键事务优先处理，并且能够保持 OCC 高吞吐量的优势。

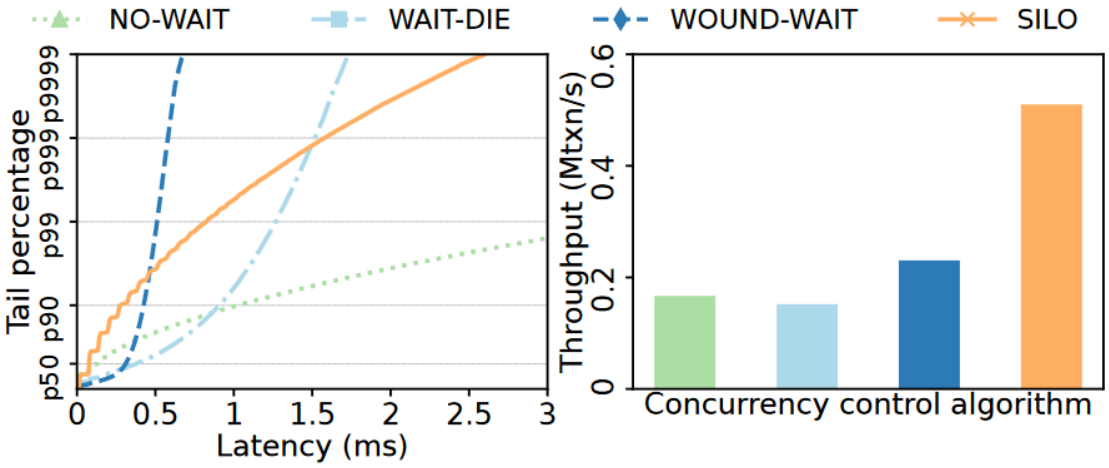


图 1. 不同并发控制方法的长尾延迟以及吞吐量，环境为 YCSB-A,r = 50%, w = 50%, $\theta$ = 0.99, 64 线程

## 3 本文方法

OCC 协议没有内置优先级的一个关键原因是，它们将冲突检测延迟到了提交阶段。虽然延迟验证能带来更高的吞吐量，但也给保证高优先级事务不被先提交的低优先级事务中止带来了挑战。

Polaris 通过在 OCC 协议中引入最低限度的悲观并发控制来解决这个问题，这样就可以使其支持优先级机制，且吞吐量不会严重下降。Polaris 引入了预定的概念：高优先级事务可以预定一条记录，这样后续的低优先级事务就无法写入这条记录；高优先级事务可以作为抢占，使先前低优先级事务的预定无效。有了预定，就能保证高优先级事务既不会被先前的低优先级事务阻止，也不会被后来的事务中止。

与悲观并发控制协议中使用的锁定机制相比，预定机制非常轻量级。首先，由于低优先级事务的读取不会导致高优先级事务的中止，因此读取永远不会被预定阻塞。其次，同一优先级的事务仍以乐观方式运行，这在很大程度上保留了 OCC 协议的高吞吐量优势。在本节的其余部分，我们将讨论如何将这一想法融入乐观并发控制协议。我们的协议主要基于 Silo，

但我们相信这一想法也可以推广到其他 OCC 协议中。我们将首先展示 Polaris 中使用的每记录数据结构，然后介绍事务处理的三个阶段。

### 3.1 TID 的组成

Silo 引入了每个记录的事务 ID（TID），其中包含一个数据版本和一个锁存位。我们对 TID 进行了扩展，增加了三个字段：优先级、优先级版本和引用计数器。为了简化讨论，我们暂时假设所有这些字段都能容纳在一个 64 位大小的空间中，这样就能用一条 CPU 指令原子式地更新。数据版本和锁存位与 Silo 中的数据版本和锁存位相同。数据版本表示记录数据是否已更新。锁存位在提交阶段的一个短窗口内保护记录数据。如果一个事务读取两次 TID，发现数据版本相同且锁存位都未被设置，那么它就能确定记录数据处于一致状态，在两次读取 TID 之间没有变化，即使 TID 的其他字段发生了变化。记录上的保留由两个 TID 字段标识：优先级和优先级版本。在 Polaris 中，一个记录可以被多个具有相同优先级但不跨优先级的事务预定。我们称这些事务为” 预定者”。TID 中的优先级字段表示预定者的优先级。通过将优先级字段重置为零 1 并递增优先级版本，可以删除记录上的保留。字段引用计数器表示访问该记录的此类被保留者的数量。它用于决定何时删除保留。在这其中，优先级为 4 位，优先级版本为 4 位，锁存位为 1 位，数据版本位为 45 位，引用计数器位为 10 位。

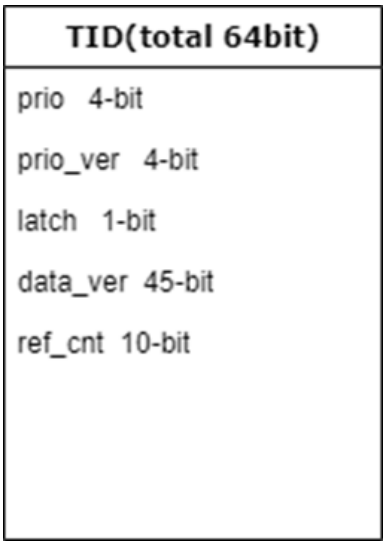


图 2. TID 的结构

### 3.2 各协议细节

#### 3.2.1 复制协议12

在事务执行过程中，需要先获取本地的副本，并同时建立该事务的读集和写集，由于在日常的事务中不存在盲写的操作，因此在写集中的记录也会出现在读集中。对于事务读取的每一条记录都会制作一份本地副本，而所有的写操作都会应用到本地中。在获取记录的复制时，需要先判断该记录是否已经被更高优先级的事务所预定，若已被预定，则副本获取失败；若引用该记录的事务优先级与当前事务相同，则该记录的引用次数加一，然后获取拷贝；若引用该记录的事务最大优先级小于当前事务，则更新该记录的优先级，并重置引用计数器，然

后获取拷贝。在能够成功获取拷贝后，需要标记是否对记录进行预定，以方便后续事务执行后的清理。

**Algorithm 1:** Record Access Protocol

```
Data: transaction priority tx.prio, record r, read-set R, write-setW, access type
      is_write
do
    do
        tid = r.tid
    while tid.latch == LOCKED;
    new_id, is_reserve = try_reserve(tid,tx.prio,is_write)
    r_local_copy = r.copy()
while !compare_and_swap(r.tid,tid,new_tid);
R.add(r,is_reserved,new_tid)
if is_write then
    W.add(r)
end
return r_local_copy
```

**Algorithm 2:** Reservation Protocol

```
Data: transaction priority tx.prio, a copy of record TID tid, access type is_write
Function try_reserve(tid, tx.prio, is_write):
    new_tid = tid
    if tid.prio == tx.prio then
        new_tid.ref++
        is_reserved = true
    else
        if tid.prio < rx.prio then
            new_tid.prio = tx.prio
            new_tid.ref_cnt = 1
            is_reserved = true
        else
            if is_write then
                ABORT()
            is_reserved = false
        end
    end
    return new_tid, is_reserved
```

### 3.2.2 提交协议3

执行结束后，事务进入提交阶段。在这个阶段，事务要验证：1）它的执行相对于其他事务是可序列化的；2）它的提交不会导致优先级更高的事务中止。对于写集中的所有记录，事务都会尝试获取其锁存器，并检查该事务的优先级是否等于或者高于这些记录的优先级，若锁存已经被设置或者该事务的优先级低于记录的优先级，则终止该事务。对于读集中的记录记录，需要验证其是否被锁定，以及数据版本是否一致，若数据版本不一致则说明有优先级更高的事务把该记录修改了。

**Algorithm 3:** Commit Protocol

**Data:** transaction priority *tx.prio*, read-set $R$, write-set $W$

```
for r in sorted(W) do
    do
        tid = r.tid
        if tid.prio > tx.prio or tid.latch == LOCKED then
            ABORT()
        locked_tid = tid locked_tid.latch = LOCKED
    while !compare_and_swap(r.tid, tid, locked_tid);
end
for r,is_reserved,tid in R do
    curr_tid = r.tid
    if curr_tid.latch == LOCKED and r not in W then
        ABORT()
    if curr_tid.data_ver != tid.data_ver then
        ABORT()
end
new_data_ver = W.max_data_ver()+1
for r in W do
    r.install_write()
    tid = r.tid
    new_tid = cleanup_write(tid, new_data_ver)
    r.tid = new_tid
end
for r, is_reserved, old_tid in R do
    if r not in W and is_reserved then
        do
            tid = r.tid
            new_tid = cleanup_read(tid, tx.priom old_tid.prio_ver)
            if new_tid == tid then
                break
        while !compare_and_swap(r.tid, tid, new_tid);
end
```

### 3.2.3 预定删除协议

在事务被终止或者该事务已经完成并成功提交后，需要对其在执行过程中所有的预定过的记录进行清理，以便后续的低优先级事务能够顺利获取。对于只读事务，判断已经预定的记录的优先级以及优先级版本与当前事务中本地副本中该记录的优先级和优先级版本一致，则把对应内存中该记录的引用次数减一，若因此减为 0，则把该记录的优先级重置为最低优先

级，并把引用次数设置为 0，同时优先级版本加一。若不一致，则无需进行任何操作。在事务被成功提交的情况下，对于写集中的记录，这些记录会被修改，需要该记录的版本进行更新，并把该记录的优先级进行重置，同时引用次数重置为 0，同时优先级版本加一。

**Algorithm 4:** Reservation Cleanup Protocol

**Data:** a copy of record TID *tid*, transaction priority *tx.prio*, previously seen priority version of the record *prio_ver*, new data version for commit *new_data_ver*

```
1  Function cleanup_read(tid, tx.prio, prio_ver):
2      if tid.latch == LOCKED then
3          return tid
4      if tid.prio != tx.prio or tid.prio_ver != prio_ver then
5          return tid
6      new_tid = tid
7      new_tid.ref_cnt−
8      if new_tid.ref_cnt == 0 then
9          new_tid.prio = 0
10         new_tid.prio_ver++
11     return new_tid
12 Function cleanup_write(tid, new_data_ver):
13     new_tid.data_ver = new_data_ver
14     new_tid.latch = UNLOCKED
15     new_tid.prio = 0
16     new_tid.prio_ver = tid.prio_ver + 1
17     new_tid.ref_cnt = 0
18     return new_tid
```

# 4 复现细节

## 4.1 与已有开源代码对比

本次论文复现内容，使用作者的源代码，在其基础上进行复现。该源代码包含了其数据的存储等功能，而本次复现的主要内容是在其上述的并发控制协议中。在复现过程中把其中与并发控制相关的文件进行删除并重写，来达到复现的目的，通过复现，加深对该优先级算法的理解。在该算法中没有加入对多次被终止多次的事务进行优先级调整的算法，在这种情况下容易导致饥饿现象的出现，因此除了对上述的协议进行复现外还加入了防止饥饿的优先级提高代码。为每个事务加上一个 abort_counter 的终止计数器, 设定事务每被中断 3 次，则优先级加 1，则优先级的计算方法如公式 1所示。

$$priority = prio_t x + (abort\_counter/3) \tag{1}$$

此外还对其中的写集排序过程进行了修改，源代码中使用了冒泡排序，虽然该排序算法易于理解，但在面对长事务时，其写集排序会占用大量的处理时间，系统会在并发控制上浪费大量时间，使得事务执行时间占比下降，因此我使用了堆排序对其进行优化。

### 4.2 实验环境搭建

实验环境使用 linux 系统，测试使用 YCSA 环境。目前大多数数据库都部署在 Linux 操作系统中，因此使用 Linux 操作系统能够更好的还原现实的使用环境。YCSA 是目前较为常用的事务处理 Benchmark，同时 YCSA 能够测量高优先级事务的延迟，能够准确的为该算法提供所需的测试结果。

## 5 实验结果分析

同时调整文件中的测试文件设置，使得程序运行本人复现的代码，获得复现代码的运行结果，然后用源代码进行一次测试，获取源代码的运行结果。通过对比两份结果可以发现，复现代码在吞吐量，延迟等方面都要略差与源代码，造成这些差异的原因可能是由于对原文的理解不够透彻，同时，在编写代码时加入了冗余的操作，导致事务在并发控制的过程中稍微逊色。此外，在实验中发现，使用堆排序和冒泡排序在事务并发处理的过程中相差并不大，这是由于，在现实生活中，一个事务即便再长，其写集也很少会去到上千条元素，而堆排序需要额外申请空间，对于一点的时间上的提升而去申请一个较大的空间，这是不划算的。


图 3. 复现代码实验结果


图 4. 源码实验结果

图 3与图 4展示了，复现代码与源代码在面对相同的测试下的测试结果吗，根据结果可以看到复现的代码与源代码之间的差距明显，在相同时间内，源代码可以处理更多的事务，也就以为着，在复现过程中存在一些代码编写的错误，导致了事务的吞吐量降低。在这样的结果下，未来还需要对代码进行进一步的改进。

## 6 总结与展望

本次针对了 Polaris 这一种事务并发控制算法进行复现，这一类型的算法的难点，并不在于如何实现该算法，而是在于如何证明该并发控制算法所能达到的并发级别，如何证明其运行后得出的结果是满足可串行化的，这是这一问题的难点。同时由于优先级在现实生活中的广泛使用，因此能不能在该算法的基础上，设计一种新的算法，在获取到事务后能够根据事务对使用者的价值，自动的生成事务的优先级，而不是需要用户手动去输入事务的优先级。这一问题，也是我正在研究的方向。

## 参考文献

[1] Ye Chenhao, Hwang Wuh-Chwen, Chen Keren, and Yu Xiangyao. Polaris: Enabling transaction priority in optimistic concurrency control. 1(1):1–24.

[2] Tu Stephen, Zheng Wenting, Kohler Eddie, Liskov Barbara, and Madden Samuel. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM.