Polaris

Enabling Transaction Priority in Optimistic Concurrency Control

Rahul Kushwaha, Jan 28, 2024

Polaris: Introduction

- Optimistic Concurrency Control protocol which supports multiple levels of priority.
- Transactions with same priority are fully optimistic.
- Prioritization is accomplished via Reservation.
- Benefits
  - Significantly lower p999 tail latency.
  - Higher throughput for high contention workloads.

Optimistic Concurrency Control

- 3 Phases of a transaction: Read, Validation, and Write.

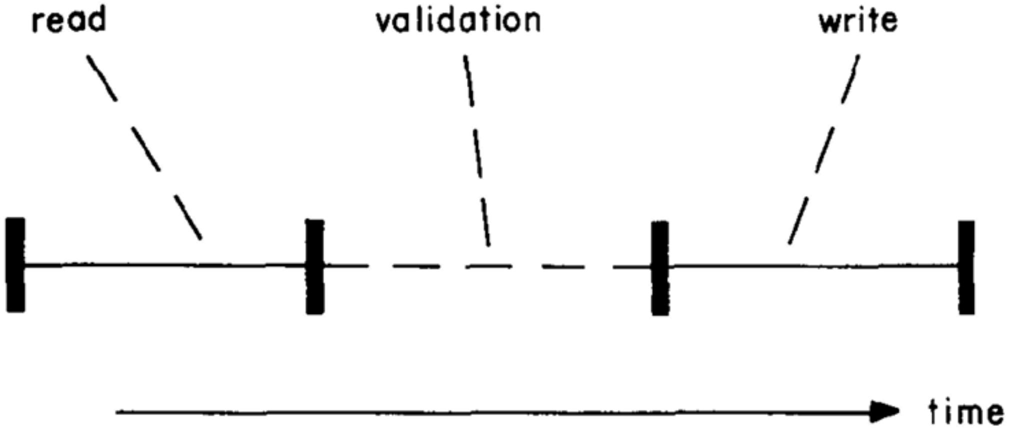


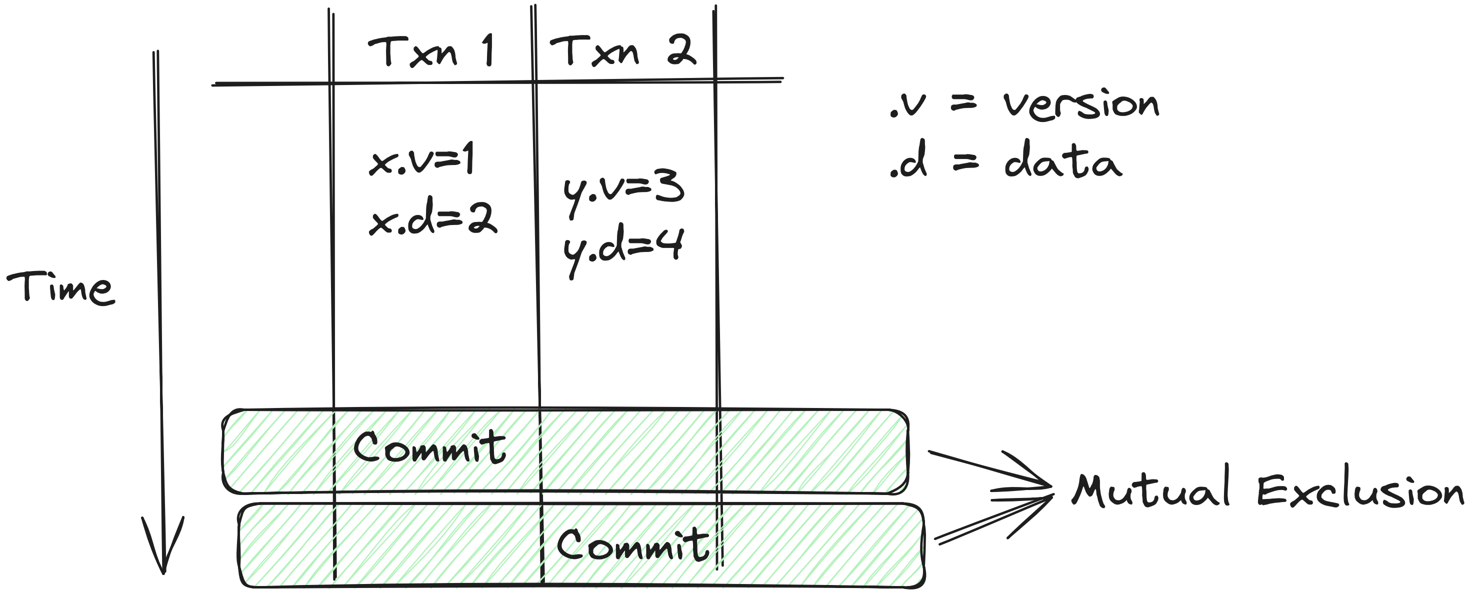
Fig. 1. The three phases of a transaction.

Img Src: [1]

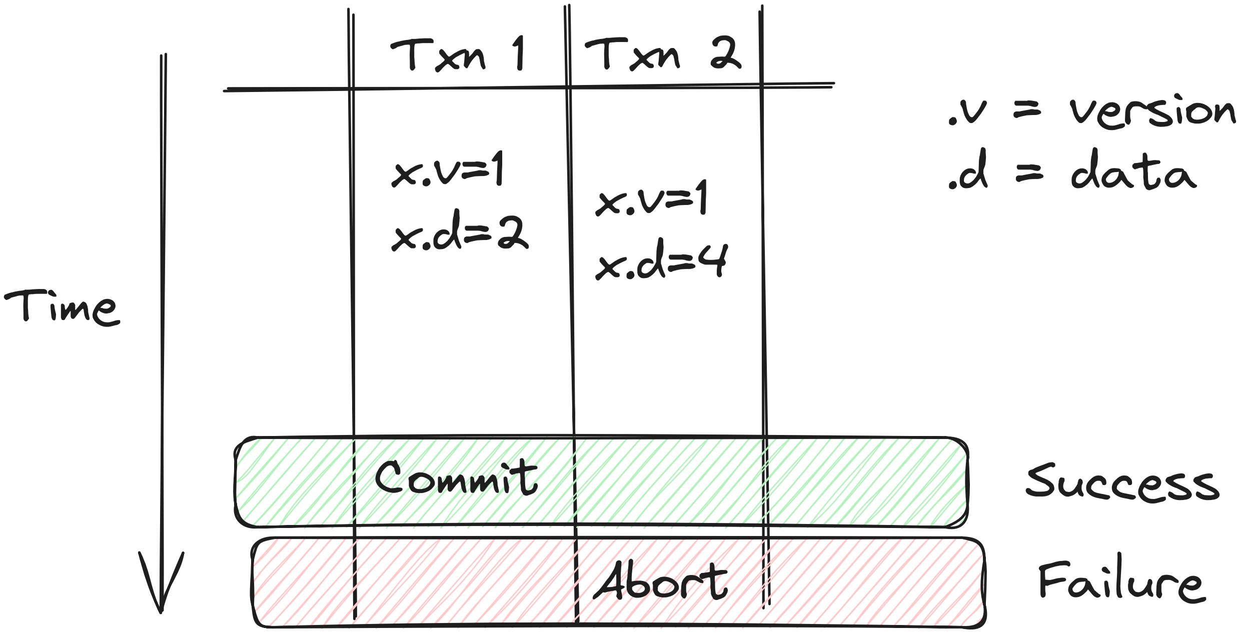
Optimistic Concurrency Control

- Read: Concurrent, multiple transactions can be executing in parallel in this phase.
  - Any mutation to the data is kept within the context of txn.
  - Read your own writes.
- Validation: Serial, global critical section.
- Write: Serial, global critical section.

Optimistic Concurrency Control



Optimistic Concurrency Control



.v = version
.d = data

Silo

- In-Memory database designed for modern multicore machines(high processor count & lots of main memory).
- Is a Serializable Database.
- Uses a variant of Optimistic Concurrency Control.
- Avoids centralized contention.
  - Example: Requires writes to shared memory only during commit phase.

Silo

Details

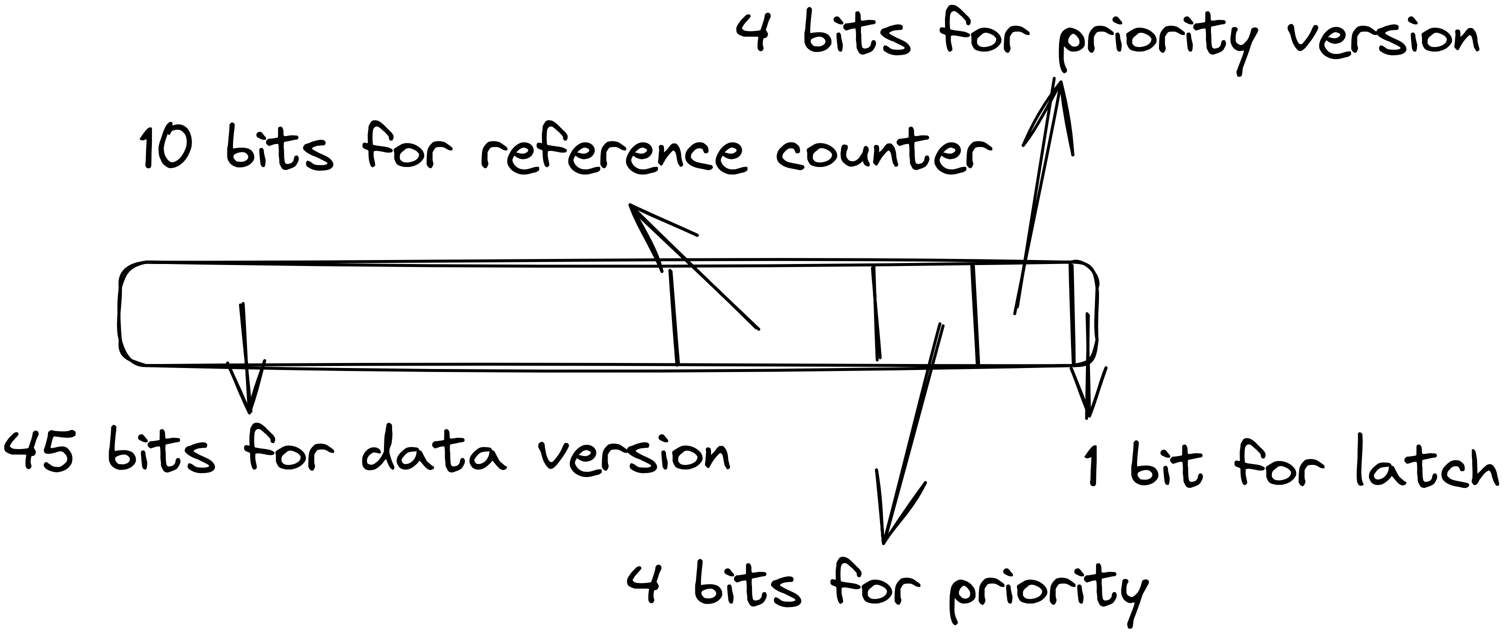
- Is based on time periods called epochs.
- Epochs form serialization points.
- A dedicated thread is responsible for perodically incrementing the epoch number(Global).
- All worker threads access Global Epoch Number during commit.

Silo

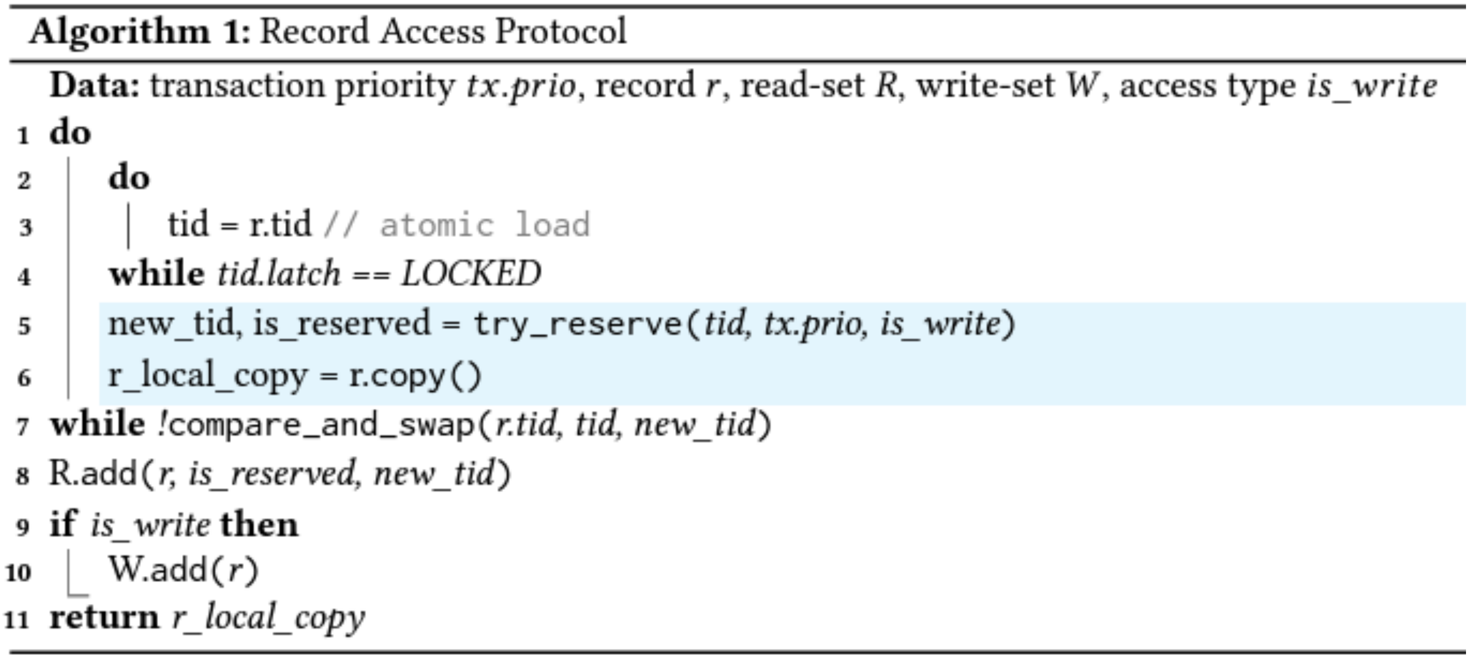
- Per-Record field, TransactionID (TID)
  - Data Version
  - Latch

Polaris TID

- Each record has a TID field.



Algorithm 1: Record Access Protocol



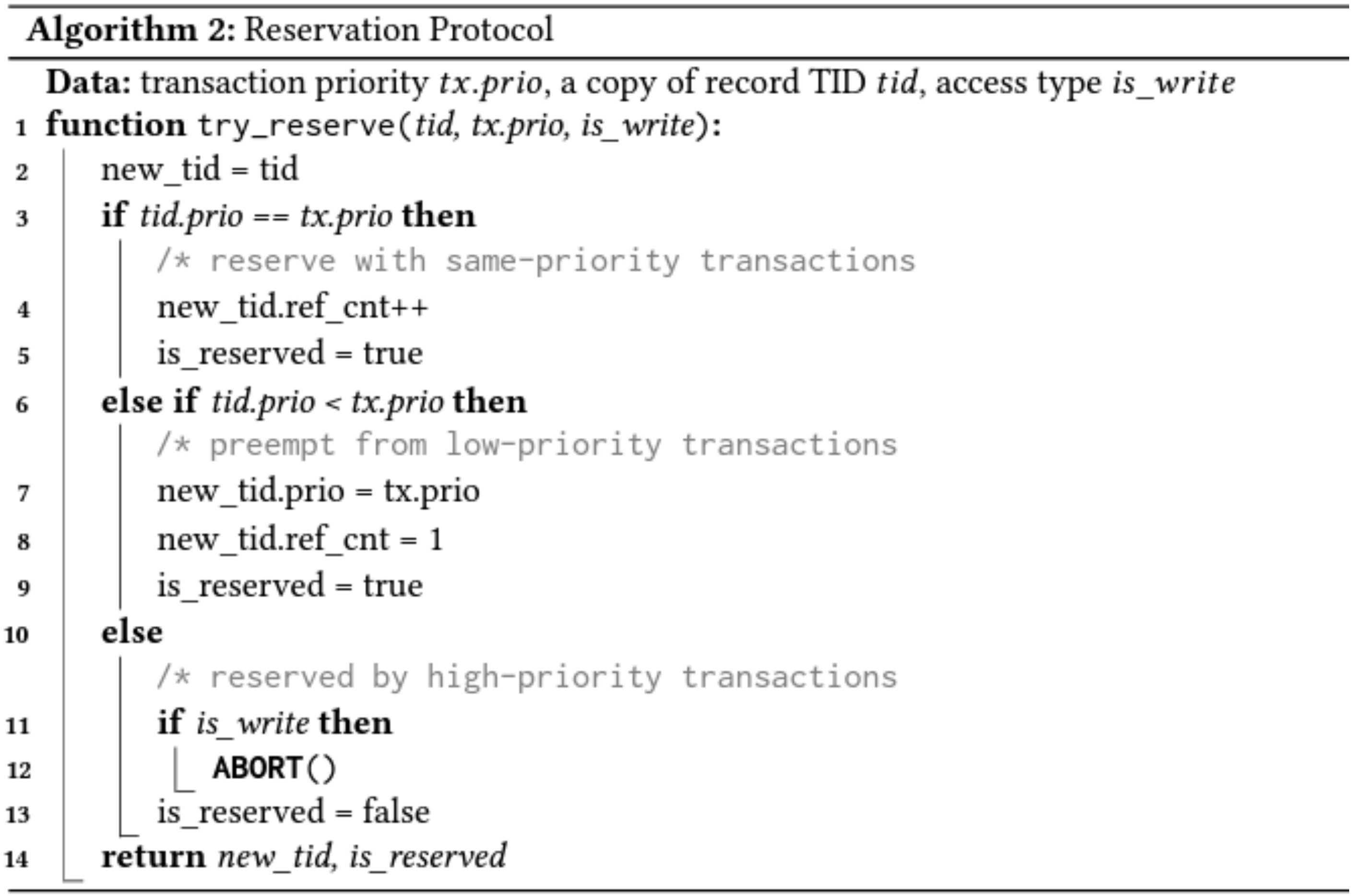
Algorithm 1: Record Access Protocol

```
Data: transaction priority tx.prio, record r, read-set R, write-set W, access type is_write
do
    do
        tid = r.tid // atomic load
    while tid.latch == LOCKED
    new_tid, is_reserved = 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)
return r_local_copy
```

Algorithm 2: Reservation Protocol

- Goal: Low-priority txn should not abort a high-priority txn.
- Example:
  - Two txns, [A -> high priority, B -> low priority]
  - If A has read the record but not committed, B should not be able to write to it, as that will cause A to abort.
  - If B has read the record but not committed, A must be able to ignore B and proceed to read/write/commit.

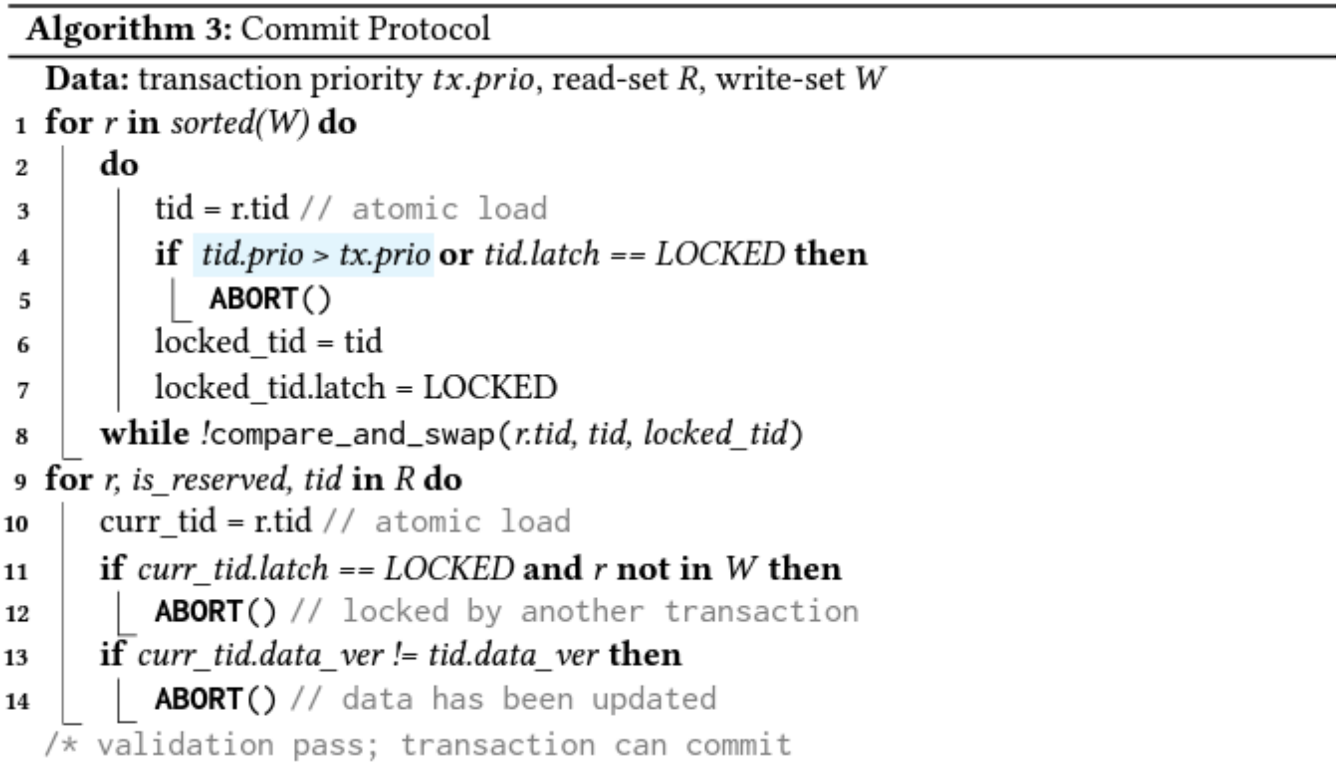
Algorithm 2: Reservation Protocol



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
        /* reserve with same-priority transactions
        new_tid.ref_cnt++
        is_reserved = true
    else if $tid.prio < tx.prio$ then
        /* preempt from low-priority transactions
        new_tid.prio = tx.prio
        new_tid.ref_cnt = 1
        is_reserved = true
    else
        /* reserved by high-priority transactions
        if $is\_write$ then
            ABORT()
        is_reserved = false
    return $new\_tid$, $is\_reserved$
```

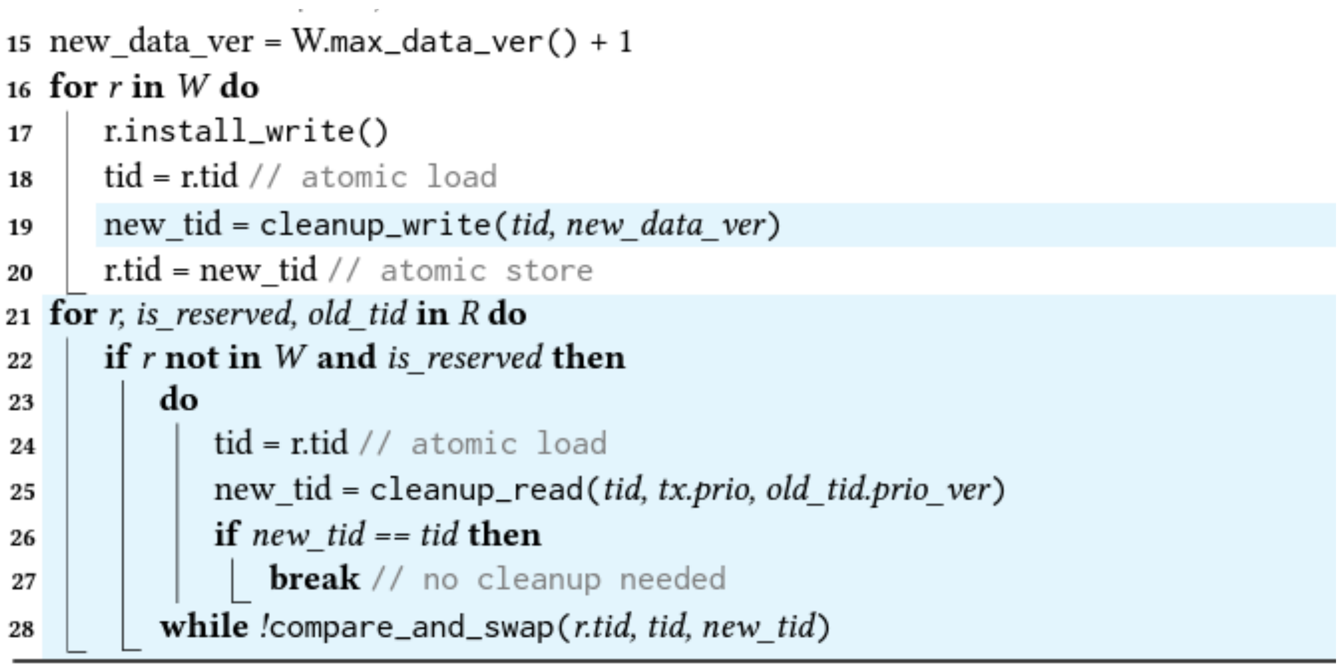
Algorithm 3: Commit Protocol(Pt-1)



Algorithm 3: Commit Protocol

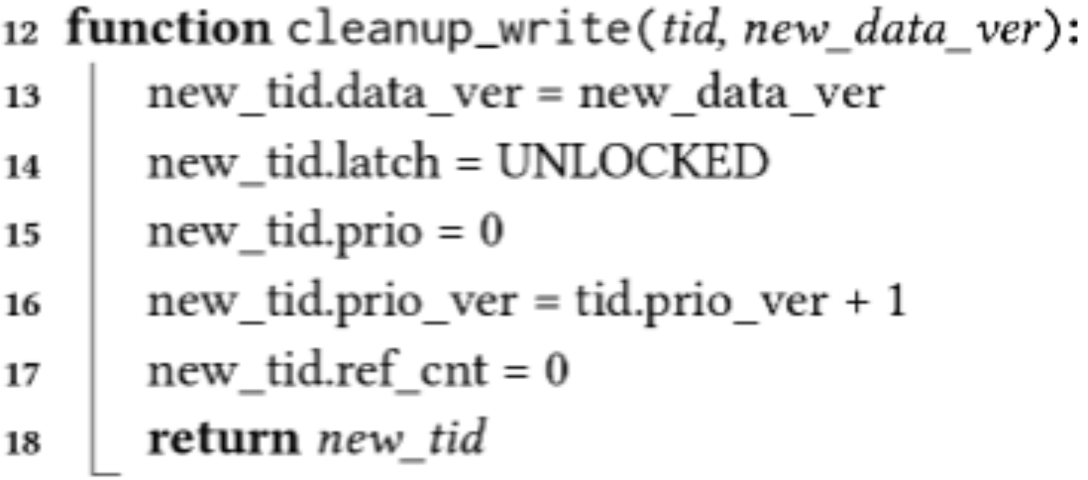
```
Data: transaction priority $tx.prio$, read-set $R$, write-set $W$
for r in sorted(W) do
    do
        tid = r.tid // atomic load
        if tid.prio > tx.prio or tid.latch == LOCKED then
            ABORT()
        locked_tid = tid
        locked_tid.latch = LOCKED
    while !compare_and_swap(r.tid, tid, locked_tid)
for r, is_reserved, tid in R do
    curr_tid = r.tid // atomic load
    if curr_tid.latch == LOCKED and r not in W then
        ABORT() // locked by another transaction
    if curr_tid.data_ver != tid.data_ver then
        ABORT() // data has been updated
/* validation pass; transaction can commit
```

Algorithm 3: Commit Protocol(Pt-2)



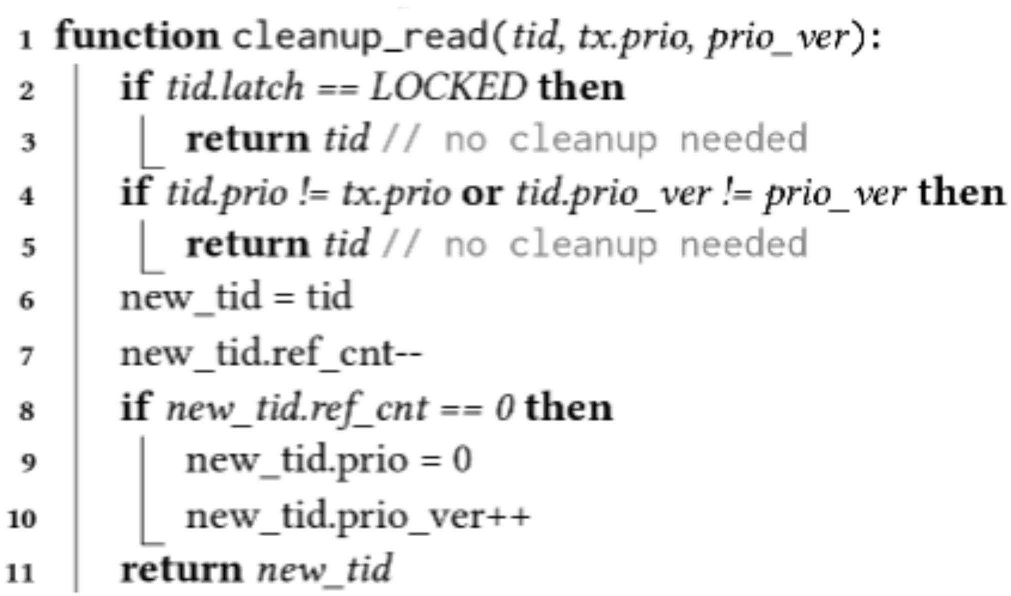
```
new_data_ver = W.max_data_ver() + 1
for r in W do
    r.install_write()
    tid = r.tid // atomic load
    new_tid = cleanup_write(tid, new_data_ver)
    r.tid = new_tid // atomic store
for r, is_reserved, old_tid in R do
    if r not in W and is_reserved then
        do
            tid = r.tid // atomic load
            new_tid = cleanup_read(tid, tx.prio, old_tid.prio_ver)
            if new_tid == tid then
                break // no cleanup needed
        while !compare_and_swap(r.tid, tid, new_tid)
```

Algorithm 4: Write CleanUp



```
function cleanup_write(tid, new_data_ver):
    new_tid.data_ver = new_data_ver
    new_tid.latch = UNLOCKED
    new_tid.prio = 0
    new_tid.prio_ver = tid.prio_ver + 1
    new_tid.ref_cnt = 0
    return new_tid
```

Algorithm 4: Read CleanUp



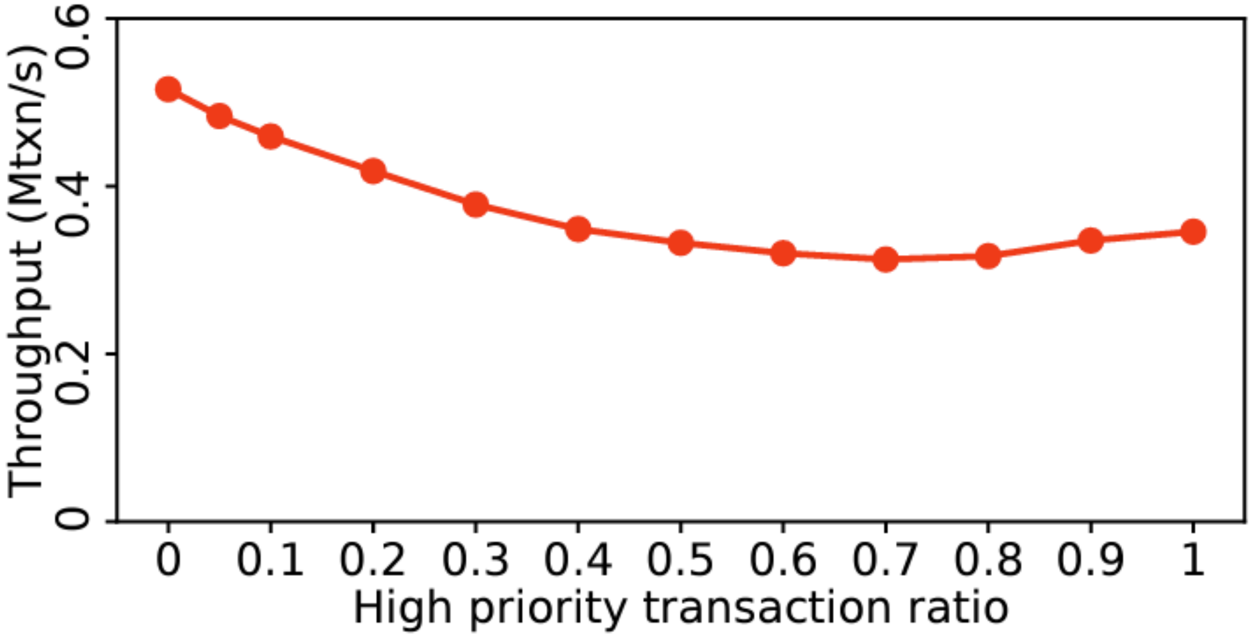
```
function cleanup_read(tid, tx.prio, prio_ver):
    if tid.latch == LOCKED then
        return tid // no cleanup needed
    if tid.prio != tx.prio or tid.prio_ver != prio_ver then
        return tid // no cleanup needed
    new_tid = tid
    new_tid.ref_cnt--
    if new_tid.ref_cnt == 0 then
        new_tid.prio = 0
        new_tid.prio_ver++
    return new_tid
```

Priority Assignment

$$p = \begin{cases} p_0, & \text{if } abort\_cnt < t \\ p_0 + \lfloor (abort\_cnt - t)/s \rfloor, & \text{otherwise} \end{cases}$$

Results: YCSB

Varying number of high priority & low priority txns



YCSB-A, r = 50%, w = 50%, θ = 0.99

Results: YCSB

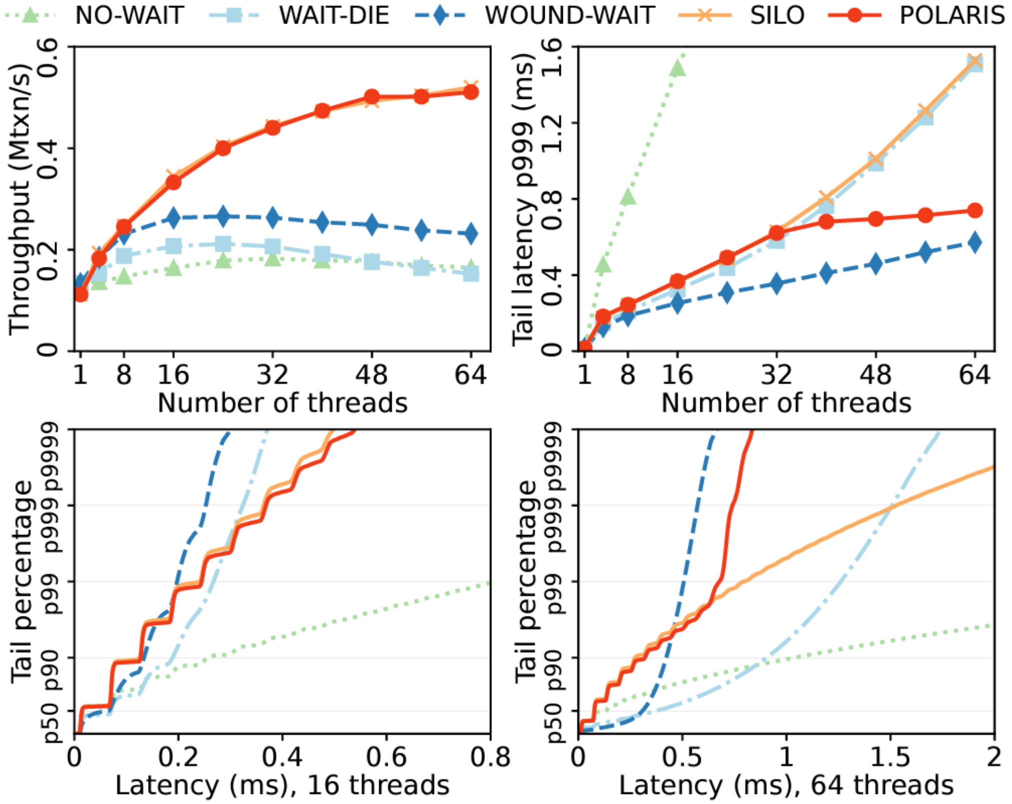
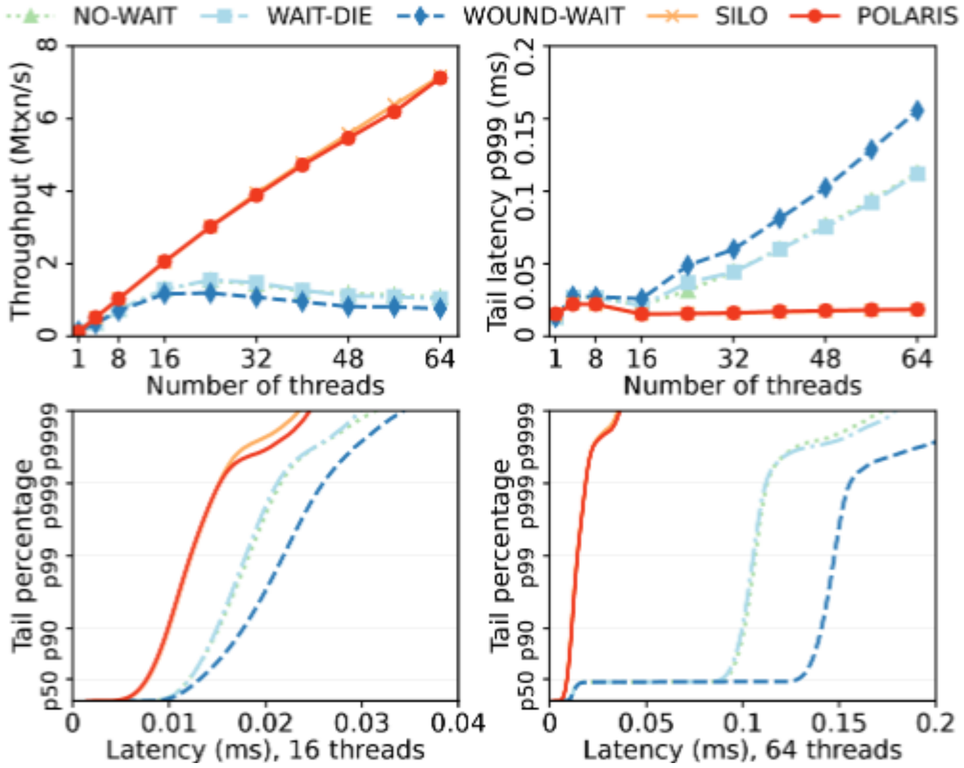


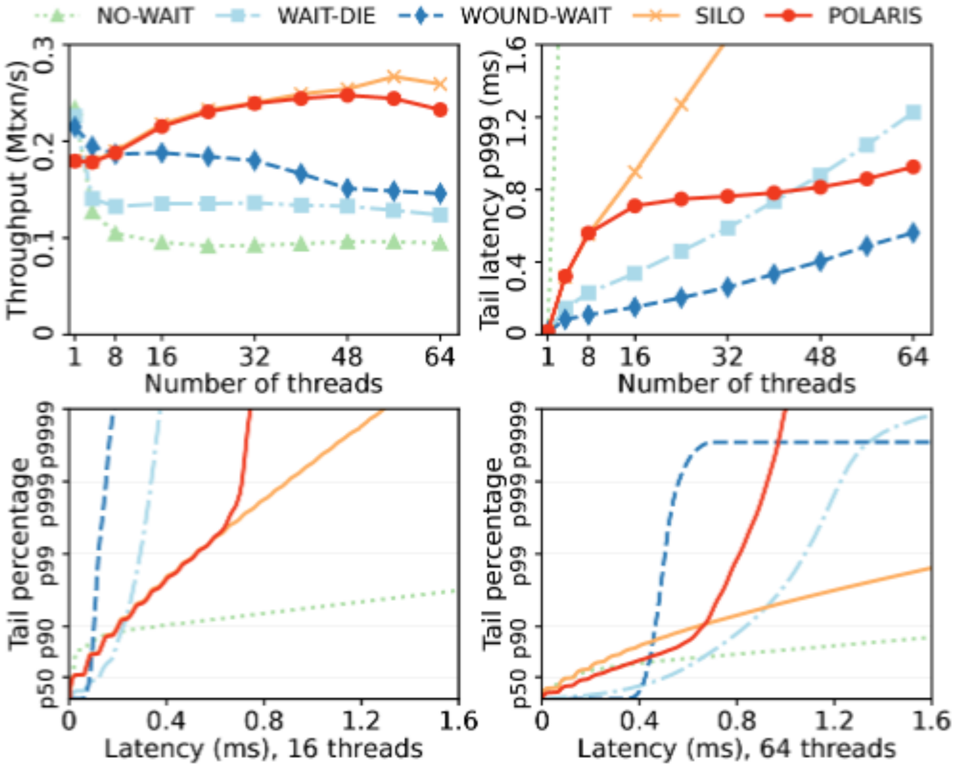
Fig. 3. Throughput and p999 tail latency over a spectrum of thread numbers; latency distribution in the cases of 16 and 64 threads (YCSB-A, $r = 50\%$, $w = 50\%$, $\theta = 0.99$).

Results: YCSB

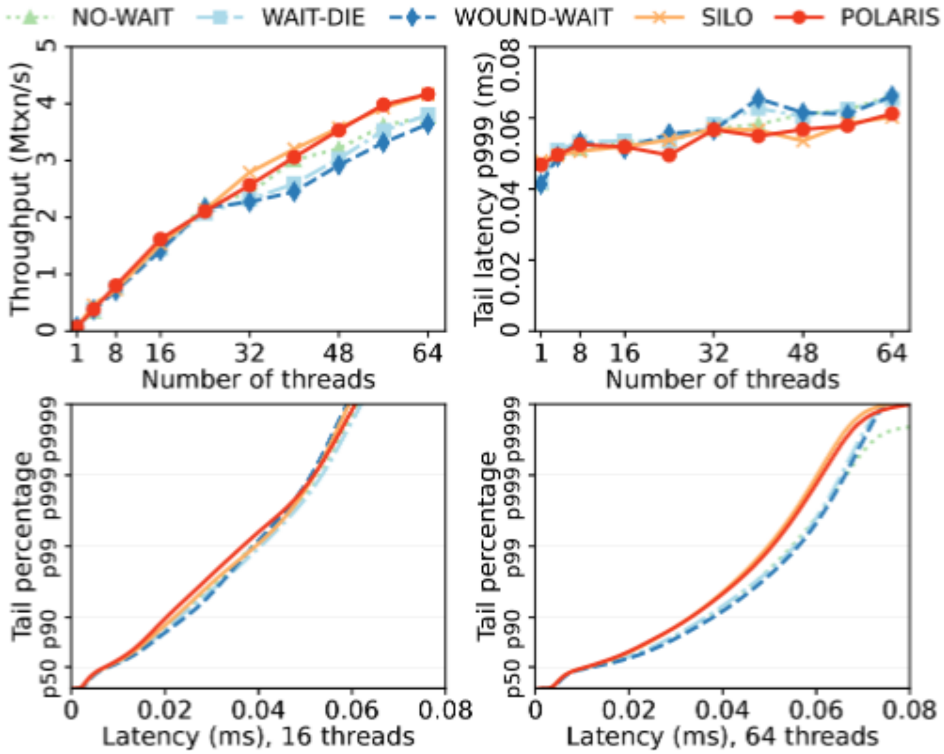


YCSB-C, r = 50%, w = 50%, θ = 0.99

Results: TPC-C [High Contention]



Results: TPC-C [Low Contention]



Thank You!!

References

- [1]H.T.Kung and JohnT.Robinson.1981. On Optimistic Methods for Concurrency Control. ACMTrans.DatabaseSyst.6, 2(jun1981),213–226. https://doi.org/10.1145/319566.319567
- [2] Stephen Tu,Wenting Zheng,Eddie Kohler,Barbara Liskov, and Samuel Madden.2013. Speedy Transactions in Multicore In-Memory Databases.In Proceedings of the Twenty-Fourth ACM Symposiumon Operating Systems Principles (Farminton,Pennsylvania)(SOSP'13).Association for Computing Machinery,NewYork,NY,USA,18–32. https://doi.org/10.1145/2517349.2522713