System Programming PROJECT2

2019040519

TaehyungKim

SSD setting

SSD – 1GB(0.75GB, 25% OP) setting

```
# Configurable SSD Controller layout parameters (must be power of 2)
secsz=512 # sector size in bytes
secs_per_pg=8 # number of sectors in a flash page
pgs_per_blk=256 # number of pages per flash block
blks_per_pl=64 # number of blocks per plane
pls_per_lun=1 # keep it at one, no multiplanes support
luns_per_ch=8 # number of chips per channel
nchs=2 # number of channels
ssd_size=768 # in megabytes, if you change the above layout parameters,
```

LPN access count code implementation

```c
uint64_t *write_lpn_access_cnt;
```

```c
void ssd_init(FemuCtrl *n)
{
    struct ssd *ssd = n->ssd;
    struct ssdparams *spp = &ssd->sp;

    ftl_assert(ssd);

    ssd_init_params(spp, n);

    /* initialize ssd internal layout architecture */
    ssd->ch = g_malloc0(sizeof(struct ssd_channel) * spp->nchs);
    for (int i = 0; i < spp->nchs; i++) {
        ssd_init_ch(&ssd->ch[i], spp);
    }

    // Allocate memory for write LPN access count array
    write_lpn_access_cnt = g_malloc0(sizeof(uint64_t) * spp->tt_pgs);

    /* initialize maptbl */
    ssd_init_maptbl(ssd);

    /* initialize rmap */
    ssd_init_rmap(ssd);

    /* initialize all the lines */
    ssd_init_lines(ssd);

    /* initialize write pointer, this is how we allocate new pages for writes */
    ssd_init_write_pointer(ssd);

    // initialize statistic buffer
    memset(stats_buffer, 0, sizeof(stats_buffer));

    qemu_thread_create(&ssd->ftl_thread, "FEMU-FTL-Thread", ftl_thread, n,
                       QEMU_THREAD_JOINABLE);
}
```

```c
static uint64_t ssd_write(struct ssd *ssd, NvmeRequest *req)
{
    uint64_t lba = req->slba;
    struct ssdparams *spp = &ssd->sp;
    int len = req->nlb;
    uint64_t start_lpn = lba / spp->secs_per_pg;
    uint64_t end_lpn = (lba + len - 1) / spp->secs_per_pg;
    struct ppa ppa;
    uint64_t lpn;
    uint64_t curlat = 0, maxlat = 0;
    int r;

    if (end_lpn >= spp->tt_pgs) {
        ftl_err("start_lpn=%" PRIu64 ",tt_pgs=%d\n", start_lpn, ssd->sp.tt_pgs);
    }

    // Update max_lpn for print lpn access count
    if (end_lpn > max_lpn)
        max_lpn = end_lpn;

    while (should_gc_high(ssd)) {
        r = do_gc(ssd, true);
        if (r == -1)
            break;
    }

    for (lpn = start_lpn; lpn <= end_lpn; lpn++) {
        ppa = get_maptbl_ent(ssd, lpn);
        if (mapped_ppa(&ppa)) {
            /* update old page information first */
            mark_page_invalid(ssd, &ppa);
            set_rmap_ent(ssd, INVALID_LPN, &ppa);
        }

        write_lpn_access_cnt[lpn]++; // increase host write lpn access count
        host_written_pages++; // increase host write page count
```

LPN access count code implementation

```c
static void *ftl_thread(void *arg)
{
    FemuCtrl *n = (FemuCtrl *)arg;
    struct ssd *ssd = n->ssd;
    NvmeRequest *req = NULL;
    uint64_t lat = 0;
    int rc;
    int i;

    while (!*(ssd->dataplane_started_ptr)) {
        usleep(100000);
    }

    /* FIXME: not safe, to handle ->to_ftl and ->to_poller gracefully */
    ssd->to_ftl = n->to_ftl;
    ssd->to_poller = n->to_poller;

    // Initialize the timer for detecting FIO and logging LPN access count
    init_timer();

    while (1) {
```

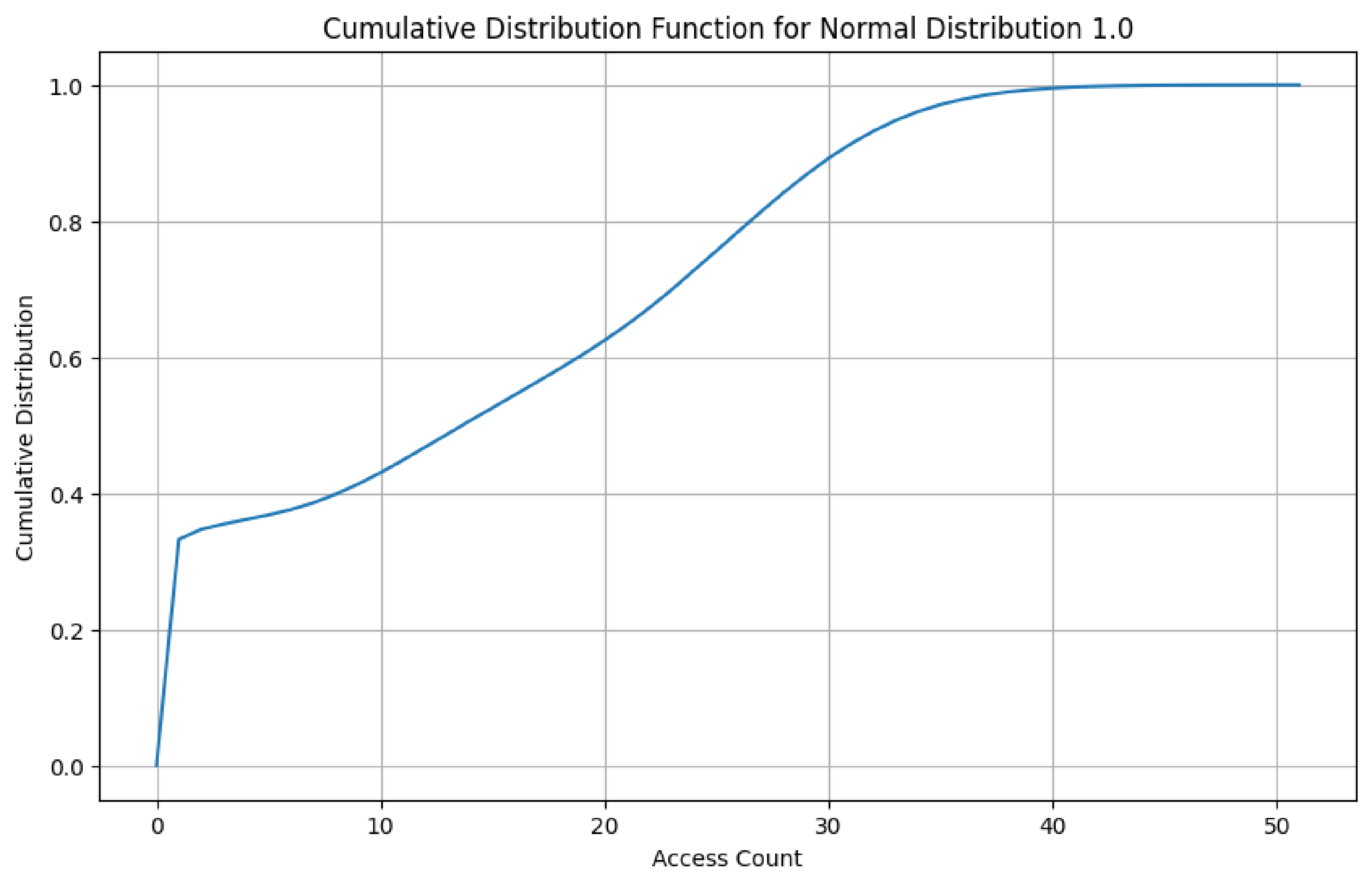
```c
void init_timer(void) {
    struct sigevent sev;
    struct itimerspec its;

    // Set up the timer to trigger a handler
    sev.sigev_notify = SIGEV_THREAD; // Notify via thread execution
    sev.sigev_value.sival_ptr = NULL;
    sev.sigev_notify_function = timer_handler; // Timer expiration handler
    sev.sigev_notify_attributes = NULL;

    // Create the timer with the specified settings
    // The timer starts in a state where it trigger the handler every 1 second
    if (timer_create(CLOCK_MONOTONIC, &sev, &timerid) == -1) {
        perror("timer_create failed");
        exit(EXIT_FAILURE);
    }

    // Set the timer's initial expriation and interval for FIO detection
    its.it_value.tv_sec = 1; // Initial expriation in 1 second
    its.it_value.tv_nsec = 0;
    its.it_interval.tv_sec = 1; // Trigger every 1 second until updated
    its.it_interval.tv_nsec = 0;

    // Start the timer with the specified settings
    if (timer_settime(timerid, TIMER_ABSTIME, &its, NULL) == -1) {
        perror("timer_settime failed");
        exit(EXIT_FAILURE);
    }
    // The timer will call timer_handler every second unitl FIO is detected.
    // Once FIO is deteceted, the interval change to 10 second.
}
```

LPN access count code implementation

```c
void timer_handler(union sigval sv) {
    static bool fio_detected = false; // Tracks if FIO workload is detected
    static time_t start_time = 0; // Start time of the FIO workload
    static int interval_count = 0; // Number of 10-second intervals processed

    // Detect FIO workload when I/O count exceeds 1000 in 1 second (initial timer interval)
    if (!fio_detected && io_cnt > 1000) {
        fio_detected = true;
        printf("FIO detected. Switching to 10-second interval logging.\n");

        // Modify the timer's expiration and interval
        struct itimerspec its;
        its.it_value.tv_sec = 10; // Set next expiration to 10 seconds
        its.it_value.tv_nsec = 0;
        its.it_interval.tv_sec = 10; // Update interval to 10 seconds
        its.it_interval.tv_nsec = 0;

        if (timer_settime(timerid, 0, &its, NULL) == -1) {
            perror("timer_settime failed");
            exit(EXIT_FAILURE);
        }
        start_time = time(NULL); // Record start time for logging
        return;
    }

    // If FIO workload is detected, log statistics every 10 seconds
    if (fio_detected) {
        ++interval_count; // Increment the number of 10-second intervals processed

        // // Record statistics in the buffer
        // Elapsed time since detecting FIO
        // stats_buffer[stats_index].seconds = difftime(time(NULL), start_time);
        // Average I/O operations per second over 10 seconds
        // stats_buffer[stats_index].iops = io_cnt / 10;
        // Write amplification factor over 10 seconds
        // stats_buffer[stats_index++].waf = (double)(host_written_pages + gc_written_pages) / host_written_pages;

        // // Reset counters for the next interval
        // io_cnt = 0;
        // host_written_pages = 0;
        // gc_written_pages = 0;

        // Stop logging after 30 intervals (300 seconds)
        if (interval_count >= 30) {
            printf("FIO finished.\n");
            print_write_lpn_access_cnt();
            // print_statistics(); // Log the recorded statistics
            timer_delete(timerid); // Delete the timer to stop further logging
        }
    }
}
```

```c
void print_write_lpn_access_cnt(void) {
    // Open a file to log lpn access count
    FILE *file = fopen("lpn_access_count.csv", "w");
    if (file == NULL) {
        perror("file open error");
        return;
    }
    // Write the header row
    fprintf(file, "LPN,Access_Count\n");
    // Log the lpn access count
    for (uint64_t lpn = 0; lpn < max_lpn; lpn++) {
        fprintf(file, "%lu,%lu\n", lpn, write_lpn_access_cnt[lpn]);
    }
    fflush(file);
    fclose(file);
}
```

Cumulative I/O Distribution Graph - Normal distribution 1.0

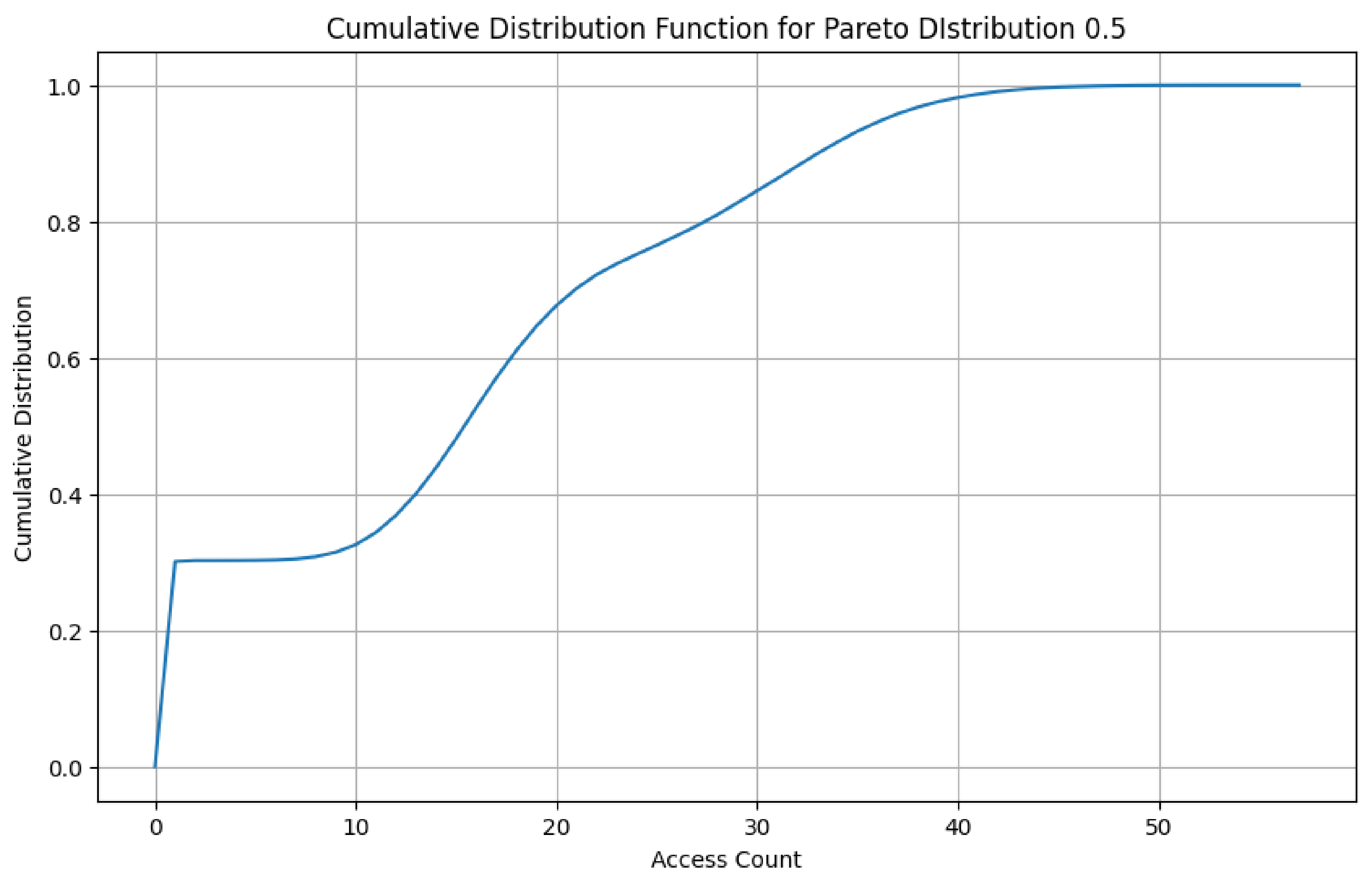


Access Count = 1 (CDF: 0.0 ~ 0.35)
Approximately 35% of LPNs have been accessed only once.

Access Count: 2 ~ 30 (CDF: 0.35 ~ 0.9)
The slope of the CDF gradually increases, indicating that a significant portion of the LPNs corresponds to the access count in this range.

Access Count: 31 ~ 51 (CDF: 0.9 ~ 1.0)
The slope of the CDF gradually decreases, indicating that only a small number of LPNs corresponds to the access count in this range.

Cumulative I/O Distribution Graph - Pareto distribution 0.5

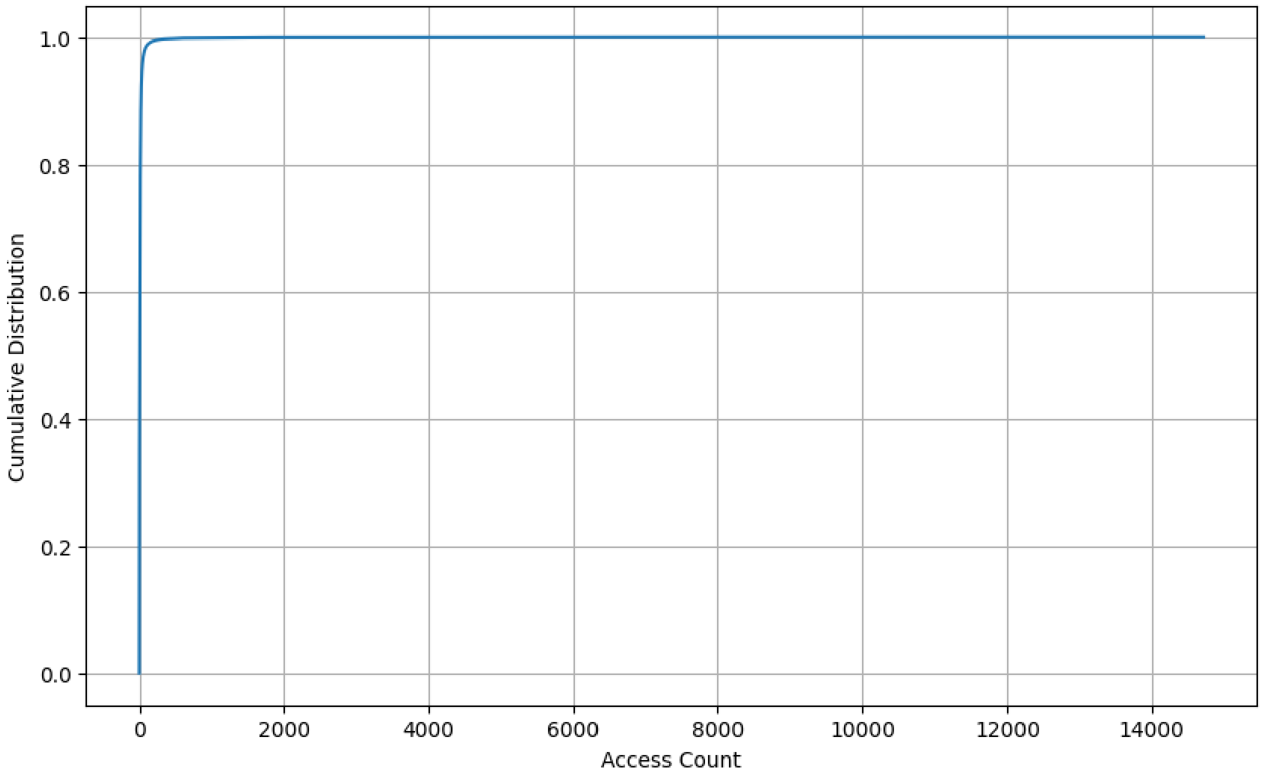


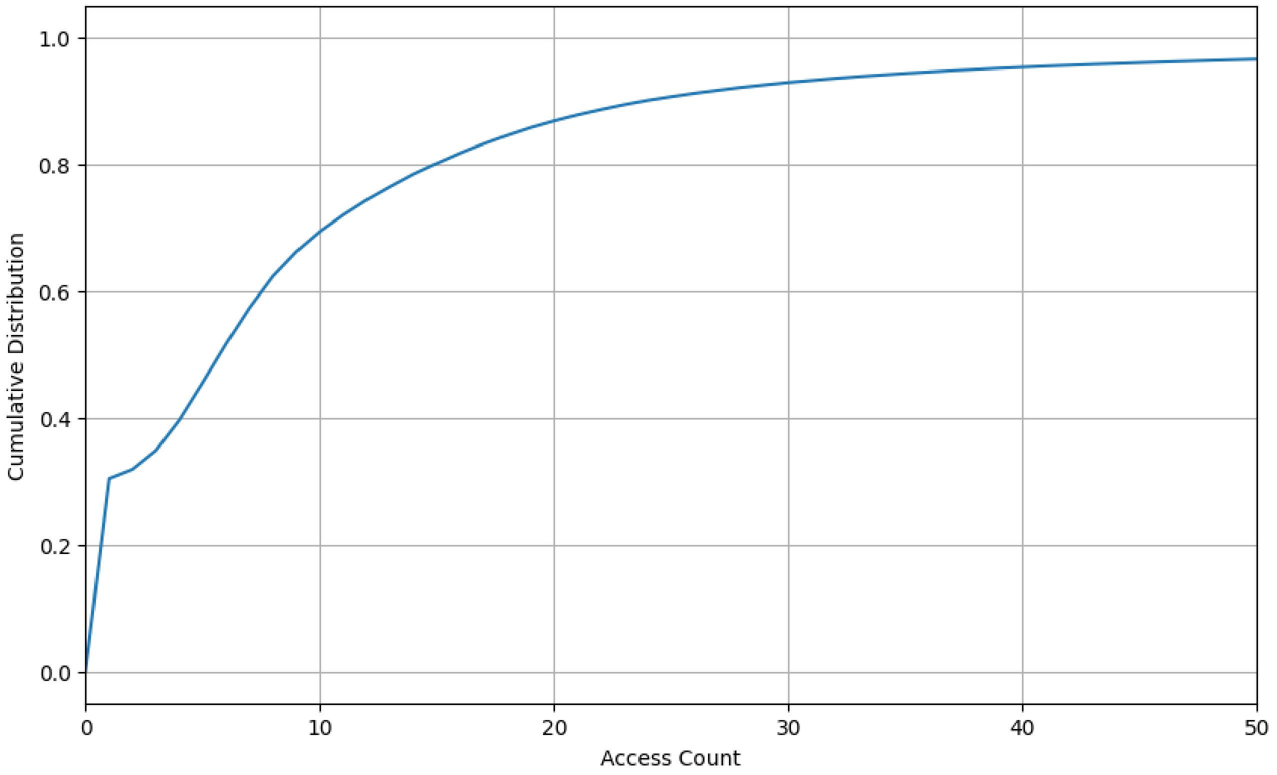
Access Count = 1 (CDF: 0.0 ~ 0.3)
Approximately 30% of LPNs have been accessed only once.

Access Count: 2 ~ 20 (CDF: 0.3 ~ 0.7)
The slope of the CDF rises sharply, indicating that a significant portion of LPNs corresponds to the access count in this range.

Access Count: 21 ~ 57 (CDF: 0.7 ~ 1.0)
The slope of the CDF gradually decreases, indicating that a smaller proportion of LPNs corresponds to the access count in this range.

Cumulative I/O Distribution Graph - Zipf distribution 0.8





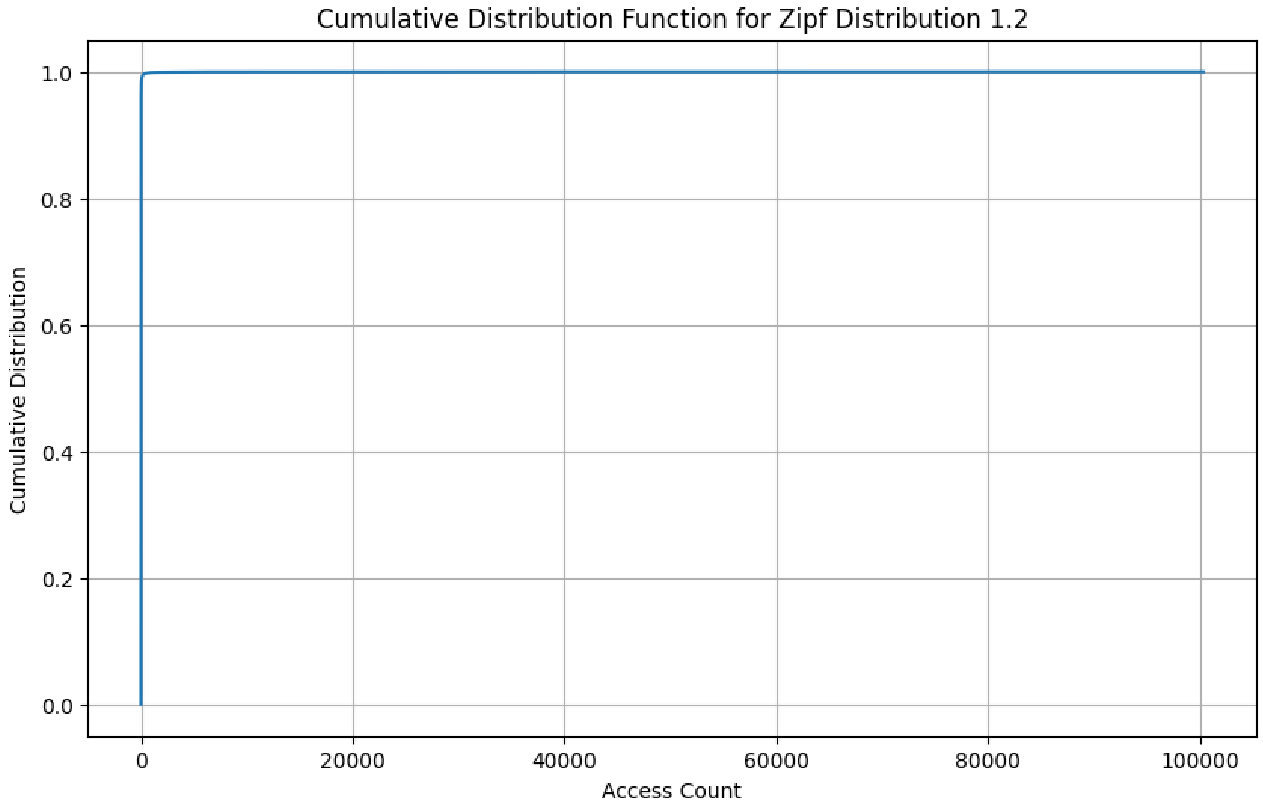
Access Count = 1 (CDF: 0.0 ~ 0.3) - Approximately 30% of LPNs have been accessed only once.

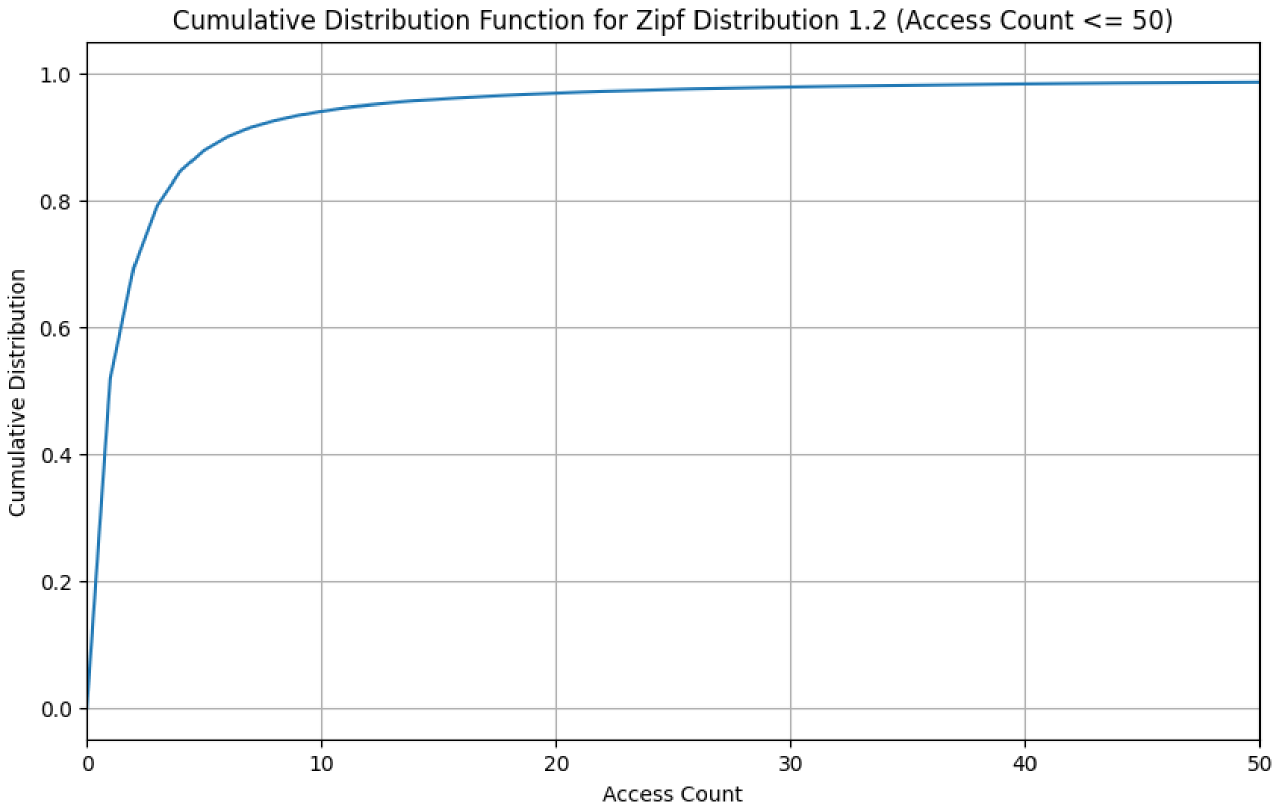
Access Count: 2 ~ 10 (CDF: 0.3 ~ 0.7) - The slope of the CDF rises sharply, indicating that a significant portion of LPNs corresponds to the access count in this range and have been accessed relatively consistently.

Access Count: 11 ~ 50 (CDF: 0.7 ~ 0.95) - The slope of the CDF decreases slightly, indicating that a smaller portion of LPNs corresponds to the access count in this range.

Access Count: 51 ~ 14590 (CDF: 0.95 ~ 1.0) - The slope of the CDF decreases dramatically, making it almost flat.
This indicates that only a small number of LPNs corresponds to the access count in this range. Notably, the access count for certain LPNs reaches as high as **14590.**

Cumulative I/O Distribution Graph - Zipf distribution 1.2





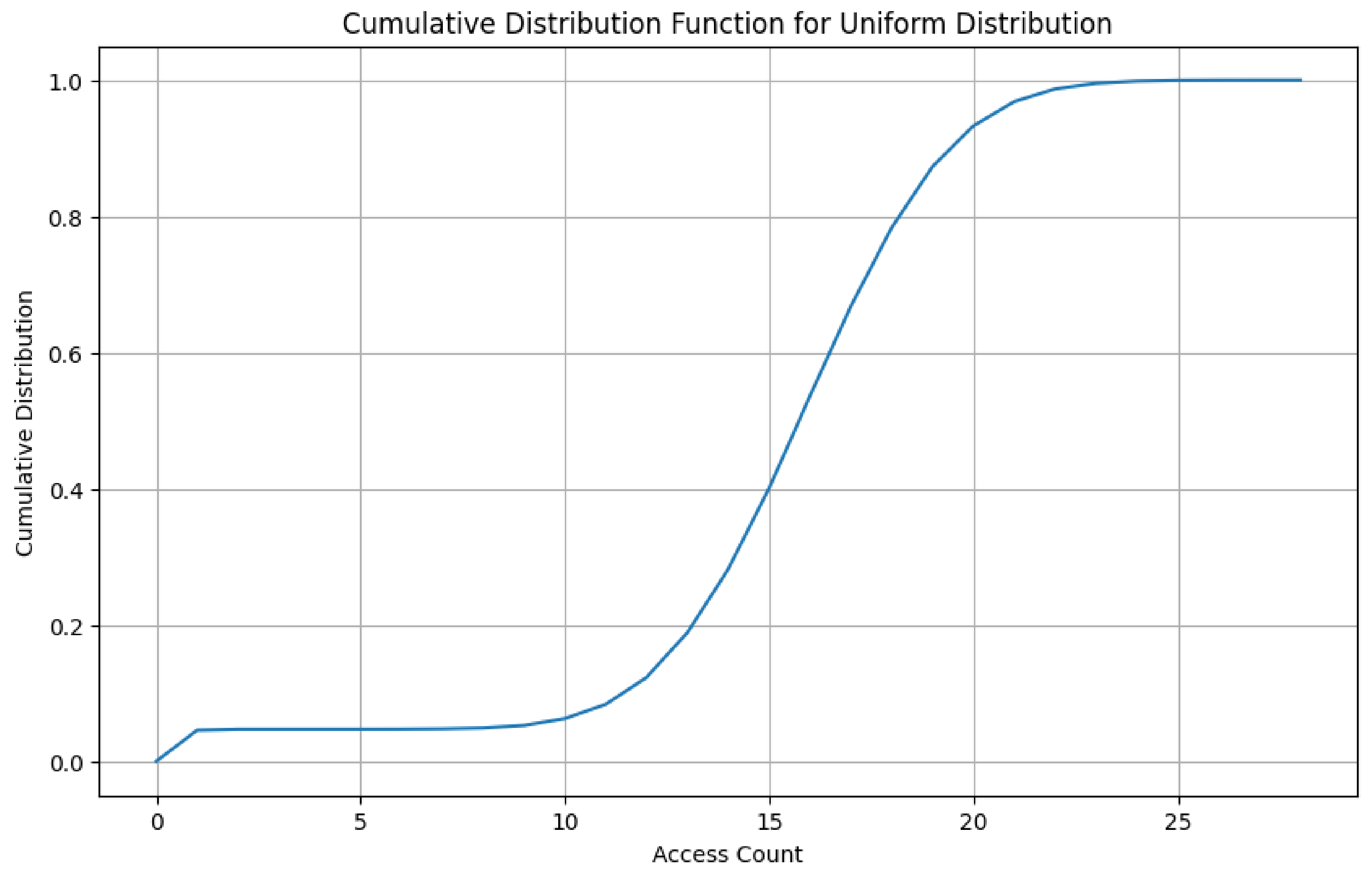
Access Count = 1 (CDF: 0.0 ~ 0.5): Approximately 50% of LPNs have been accessed only once.

Access Count: 2 ~ 6 (CDF: 0.5 ~ 0.9): The slope of the CDF rises sharply, indicating that a significant portion of LPNs corresponds to the access count in this range.

Access Count: 7 ~ 50 (CDF: 0.9 ~ 0.99): The slope of the CDF decreases slightly, indicating that a smaller portion of LPNs corresponds to the access count in this range

Access Count: 51 ~ 112023 (CDF: 0.99 ~ 1.0): The slope of the CDF decreases dramatically, making it almost flat. This indicates that only a small number of LPNs corresponds to the access count in this range. Notably, the access count for certain LPNs reaches as high as **112023**.

Cumulative I/O Distribution Graph - Uniform distribution



Access Count = 1 (CDF: 0.0 ~ 0.05)
Approximately 5% of LPNs have been accessed only once.

Access Count: 2 ~ 18 (CDF: 0.05 ~ 0.9)
The slope of the CDF gradually increases and then stabilizes, indicating that a significant portion of LPNs corresponds to the access counts in this range, which are relatively balanced.

Access Count: 19 ~ 36 (CDF: 0.9 ~ 1.0)
The slope of the CDF gradually decreases, suggesting that only a small portion of LPNs corresponds to the access counts in this range.

Hot/cold separation code implementation

```
#define BUFFER_SIZE 30
#define HOT_THRESHOLD 10

uint64_t *write_lpn_access_cnt;
uint64_t io_cnt = 0;
uint64_t host_written_pages = 0;
uint64_t gc_written_pages = 0;

static stats stats_buffer[BUFFER_SIZE];
static size_t stats_index = 0;
static timer_t timerid;
uint64_t max_lpn = 0;
```

```
// Structure to hold performance statistics
typedef struct {
    double seconds;
    uint64_t iops;
    double waf;
} stats;
```

```
struct ssd {
    char *ssdname;
    struct ssdparams sp;
    struct ssd_channel *ch;
    struct ppa *maptbl; /* page level mapping table */
    uint64_t *rmap;     /* reverse mapptbl, assume it's stored in OOB */
    struct write_pointer cp; // cold data write pointer
    struct write_pointer hp; // hot data write pointer
    struct line_mgmt lm;

    /* lockless ring for communication with NVMe IO thread */
    struct rte_ring **to_ftl;
    struct rte_ring **to_poller;
    bool *dataplane_started_ptr;
    QemuThread ftl_thread;
};
```

```
// Determine the type of data (hot or cold) based on LPN access count
static inline bool get_data_type(uint64_t lpn) {
    // If the write access count for the given LPN exceeds the hot threshold,
    // classify the data as "hot" (return true). Otherwise, classify it as "cold" (return false).
    return write_lpn_access_cnt[lpn] > HOT_THRESHOLD;
}
```

```
static void ssd_init_write_pointer(struct ssd *ssd)
{
    // initialize hot pointer
    struct write_pointer *hp = &ssd->hp;
    struct line_mgmt *lm = &ssd->lm;
    struct line *curline = NULL;

    curline = QTAILQ_FIRST(&lm->free_line_list);
    QTAILQ_REMOVE(&lm->free_line_list, curline, entry);
    lm->free_line_cnt--;
    /* wpp->curline is always our next-to-write super-block */

    hp->curline = curline;
    hp->ch = 0;
    hp->lun = 0;
    hp->pg = 0;
    // 'Line' groups blocks with the same number across chips.
    hp->blk = curline->id;
    hp->pl = 0;

    // initalize cold pointer
    struct write_pointer *cp = &ssd->cp;
    curline = QTAILQ_FIRST(&lm->free_line_list);
    QTAILQ_REMOVE(&lm->free_line_list, curline, entry);
    lm->free_line_cnt--;
    /* wpp->curline is always our next-to-write super-block */

    cp->curline = curline;
    cp->ch = 0;
    cp->lun = 0;
    cp->pg = 0;
    // same logic applies for cold pointer
    cp->blk = curline->id;
    cp->pl = 0;
}
```

Hot/cold separation code implementation

```c
static struct ppa get_new_page(struct ssd *ssd, bool type)
{
    // Determine which write pointer to use based on the type
    // If 'type' is true, use the hot write pointer (hp)
    // If 'type' is false, use the cold write pointer (cp)
    struct write_pointer *wpp = type ? &ssd->hp : &ssd->cp;

    struct ppa ppa;
    ppa.ppa = 0;
    ppa.g.ch = wpp->ch;
    ppa.g.lun = wpp->lun;
    ppa.g.pg = wpp->pg;
    ppa.g.blk = wpp->blk;
    ppa.g.pl = wpp->pl;
    ftl_assert(ppa.g.pl == 0);

    return ppa;
}
```

```c
static void ssd_advance_write_pointer(struct ssd *ssd, bool type)
{
    struct ssdparams *spp = &ssd->sp;
    // Determine which write pointer to use based on the type
    // If 'type' is true, use the hot write pointer (hp)
    // If 'type' is false, use the cold write pointer (cp)
    struct write_pointer *wpp = type ? &ssd->hp : &ssd->cp;
    struct line_mgmt *lm = &ssd->lm;

    check_addr(wpp->ch, spp->nchs);
    wpp->ch++;
    if (wpp->ch == spp->nchs) {
        wpp->ch = 0;
        check_addr(wpp->lun, spp->luns_per_ch);
        wpp->lun++;
        /* in this case, we should go to next lun */
        if (wpp->lun == spp->luns_per_ch) {
            wpp->lun = 0;
            /* go to next page in the block */
            check_addr(wpp->pg, spp->pgs_per_blk);
            wpp->pg++;
```

Hot/cold separation code implementation

```c
static uint64_t ssd_write(struct ssd *ssd, NvmeRequest *req)
{
    uint64_t lba = req->slba;
    struct ssdparams *spp = &ssd->sp;
    int len = req->nlb;
    uint64_t start_lpn = lba / spp->secs_per_pg;
    uint64_t end_lpn = (lba + len - 1) / spp->secs_per_pg;
    struct ppa ppa;
    uint64_t lpn;
    uint64_t curlat = 0, maxlat = 0;
    int r;

    if (end_lpn >= spp->tt_pgs) {
        ftl_err("start_lpn=%" PRIu64 ",tt_pgs=%d\n", start_lpn, ssd->sp.tt_pgs);
    }

    // Update max_lpn for print lpn access count
    if (end_lpn > max_lpn)
        max_lpn = end_lpn;

    while (should_gc_high(ssd)) {
        r = do_gc(ssd, true);
        if (r == -1)
            break;
    }

    for (lpn = start_lpn; lpn <= end_lpn; lpn++) {
        ppa = get_maptbl_ent(ssd, lpn);
        if (mapped_ppa(&ppa)) {
            /* update old page information first */
            mark_page_invalid(ssd, &ppa);
            set_rmap_ent(ssd, INVALID_LPN, &ppa);
        }

        write_lpn_access_cnt[lpn]++; // increase host write lpn access count
        host_written_pages++; // increase host write page count
```

```c
        // Determine the data type (hot or cold) based on the write lpn access count
        bool type = get_data_type(lpn);
        // Allocate a new page for the corresponding type (hot or cold)
        ppa = get_new_page(ssd, type);

        /* update maptbl */
        set_maptbl_ent(ssd, lpn, &ppa);

        /* update rmap */
        set_rmap_ent(ssd, lpn, &ppa);

        mark_page_valid(ssd, &ppa);

        // Advance the write pointer for the corresponding type (hot or cold)
        ssd_advance_write_pointer(ssd, type);

        struct nand_cmd swr;
        swr.type = USER_IO;
        swr.cmd = NAND_WRITE;
        swr.stime = req->stime;
        /* get latency statistics */
        curlat = ssd_advance_status(ssd, &ppa, &swr);
        maxlat = (curlat > maxlat) ? curlat : maxlat;
    }
    return maxlat;
}
```

Hot/cold separation code implementation

```c
/* here ppa identifies the block we want to clean */
static void clean_one_block(struct ssd *ssd, struct ppa *ppa)
{
    struct ssdparams *spp = &ssd->sp;
    struct nand_page *pg_iter = NULL;
    int cnt = 0;

    for (int pg = 0; pg < spp->pgs_per_blk; pg++) {
        ppa->g.pg = pg;
        pg_iter = get_pg(ssd, ppa);
        /* there shouldn't be any free page in victim blocks */
        ftl_assert(pg_iter->status != PG_FREE);
        if (pg_iter->status == PG_VALID) {
            gc_read_page(ssd, ppa);
            /* delay the maptbl update until "write" happens */
            gc_write_page(ssd, ppa);
            gc_written_pages++; // increase gc written pages count
            cnt++;
        }
    }

    ftl_assert(get_blk(ssd, ppa)->vpc == cnt);
}
```

```c
/* move valid page data (already in DRAM) from victim line to a new page */
static uint64_t gc_write_page(struct ssd *ssd, struct ppa *old_ppa)
{
    struct ppa new_ppa;
    struct nand_lun *new_lun;

    uint64_t lpn = get_rmap_ent(ssd, old_ppa);

    ftl_assert(valid_lpn(ssd, lpn));
    // Determine the data type (hot or cold) based on the write lpn access count

    bool type = get_data_type(lpn);
    // Allocate a new page for the corresponding type (hot or cold)
    new_ppa = get_new_page(ssd, type);

    /* update maptbl */
    set_maptbl_ent(ssd, lpn, &new_ppa);
    /* update rmap */
    set_rmap_ent(ssd, lpn, &new_ppa);

    mark_page_valid(ssd, &new_ppa);

    // Advance the write pointer for the corresponding type (hot or cold)
    ssd_advance_write_pointer(ssd, type);
```

Hot/cold separation code implementation

```c
static void *ftl_thread(void *arg)
{
    FemuCtrl *n = (FemuCtrl *)arg;
    struct ssd *ssd = n->ssd;
    NvmeRequest *req = NULL;
    uint64_t lat = 0;
    int rc;
    int i;

    while (!*(ssd->dataplane_started_ptr)) {
        usleep(100000);
    }

    /* FIXME: not safe, to handle ->to_ftl and ->to_poller gracefully */
    ssd->to_ftl = n->to_ftl;
    ssd->to_poller = n->to_poller;

    // Initialization for FIO detection and statistics
    init_timer();

    while (1) {
        for (i = 1; i <= n->nr_pollers; i++) {
            if (!ssd->to_ftl[i] || !femu_ring_count(ssd->to_ftl[i]))
                continue;

            rc = femu_ring_dequeue(ssd->to_ftl[i], (void *)&req, 1);
            if (rc != 1) {
                printf("FEMU: FTL to_ftl dequeue failed\n");
            }

            ftl_assert(req);
            switch (req->cmd.opcode) {
            case NVME_CMD_WRITE:
                lat = ssd_write(ssd, req);
                io_cnt++;
                break;
            case NVME_CMD_READ:
                lat = ssd_read(ssd, req);
                io_cnt++;
                break;
            case NVME_CMD_DSM:
                lat = 0;
                break;
            default:
                // ftl_err("FTL received unkown request type, ERROR\n");
                ;
            }
```

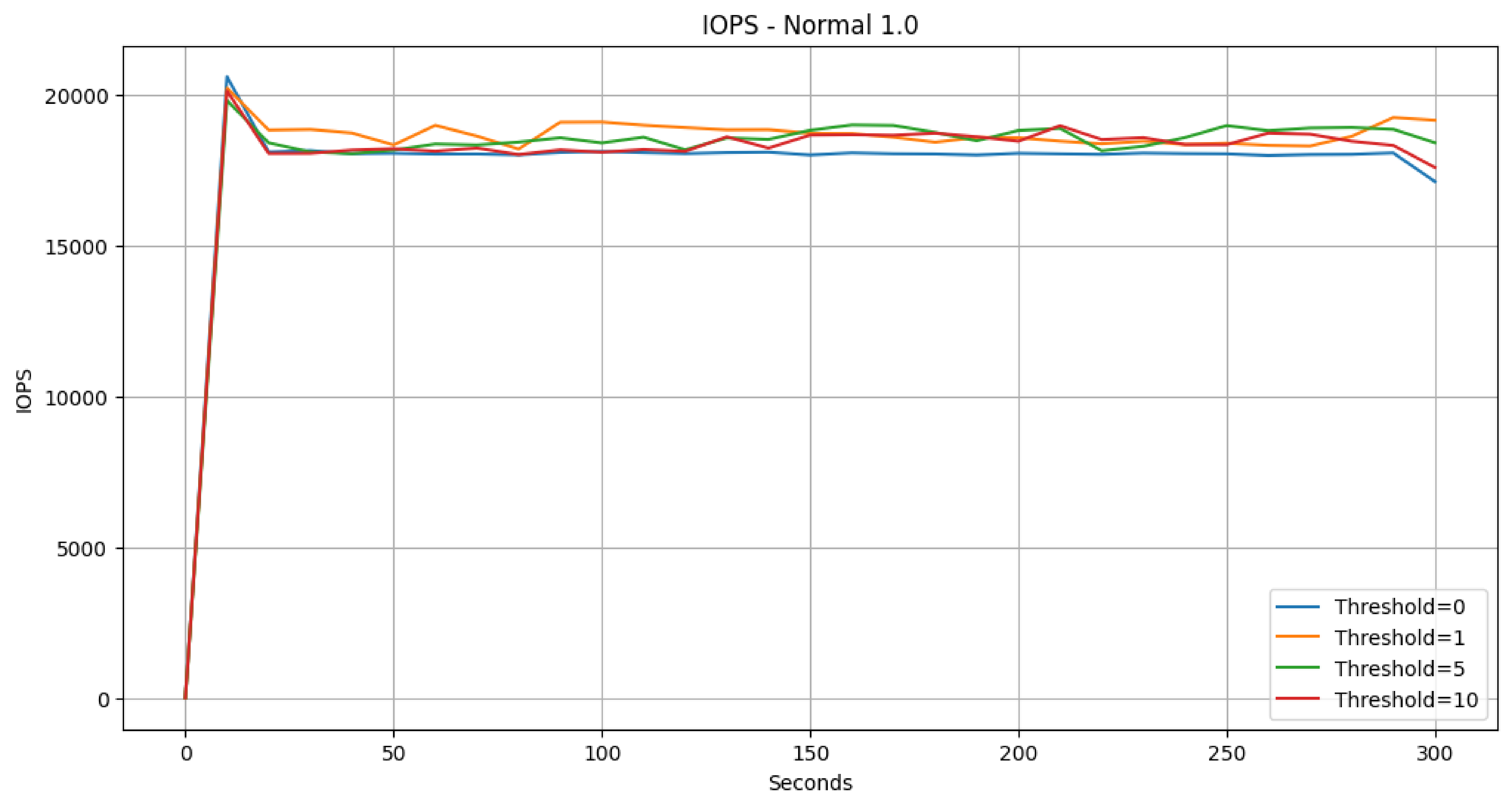
```c
void init_timer(void) {
    struct sigevent sev;
    struct itimerspec its;

    // Set up the timer to trigger a handler
    sev.sigev_notify = SIGEV_THREAD; // Notify via thread execution
    sev.sigev_value.sival_ptr = NULL;
    sev.sigev_notify_function = timer_handler; // Timer expiration handler
    sev.sigev_notify_attributes = NULL;

    // Create the timer with the specified settings
    // The timer starts in a state where it trigger the handler every 1 second
    if (timer_create(CLOCK_MONOTONIC, &sev, &timerid) == -1) {
        perror("timer_create failed");
        exit(EXIT_FAILURE);
    }

    // Set the timer's initial expriation and interval for FIO detection
    its.it_value.tv_sec = 1; // Initial expriation in 1 second
    its.it_value.tv_nsec = 0;
    its.it_interval.tv_sec = 1; // Trigger every 1 second until updated
    its.it_interval.tv_nsec = 0;

    // Start the timer with the specified settings
    if (timer_settime(timerid, TIMER_ABSTIME, &its, NULL) == -1) {
        perror("timer_settime failed");
        exit(EXIT_FAILURE);
    }
    // The timer will call timer_handler every second unitl FIO is detected.
    // Once FIO is detecteted, the interval change to 10 second.
}
```

Hot/cold separation code implementation

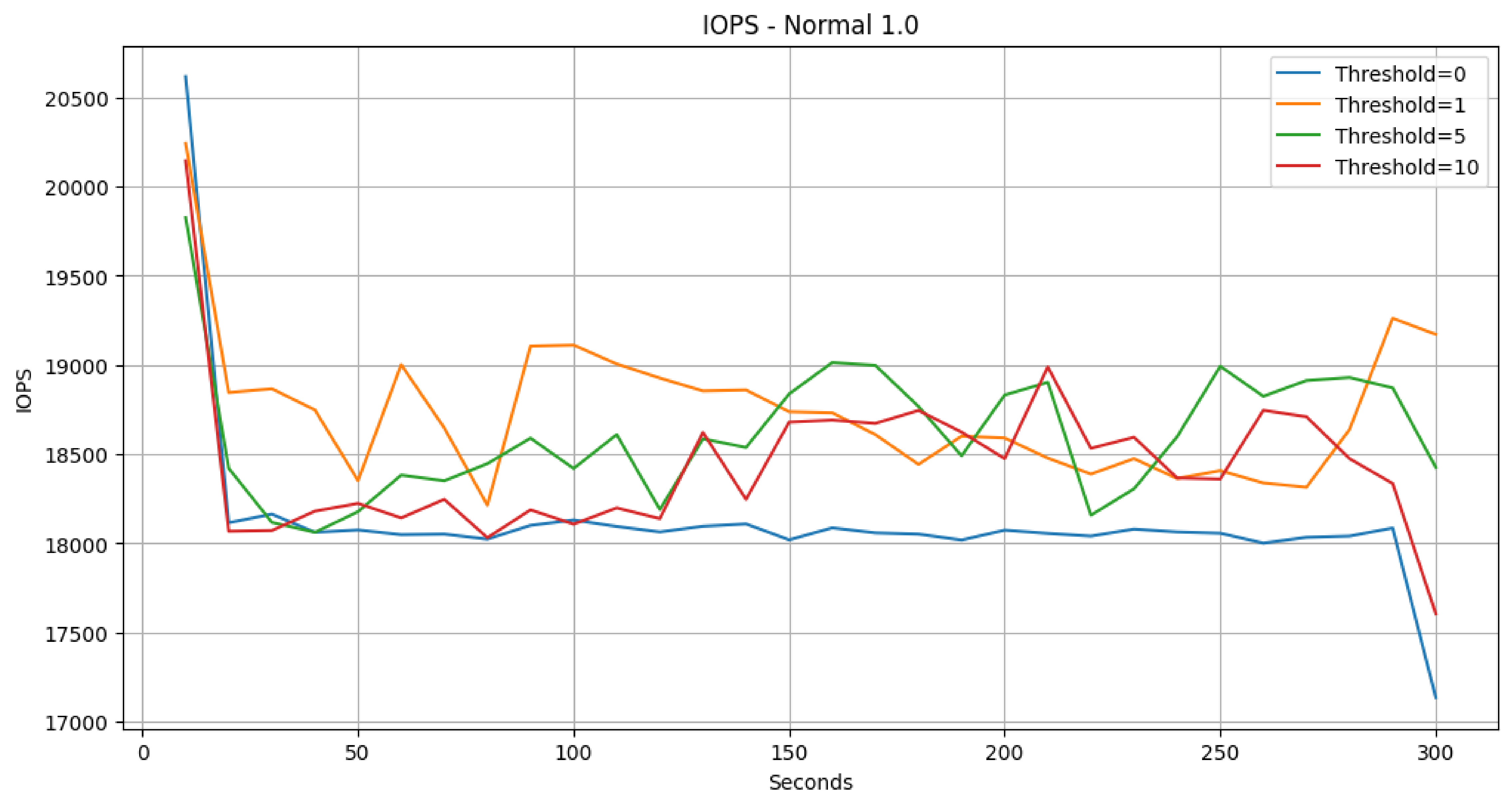
```c
void timer_handler(union sigval sv) {
    static bool fio_detected = false; // Tracks if FIO workload is detected
    static time_t start_time = 0; // Start time of the FIO workload
    static int interval_count = 0; // Number of 10-second intervals processed

    // Detect FIO workload when I/O count exceeds 1000 in 1 second (initial timer interval)
    if (!fio_detected && io_cnt > 1000) {
        fio_detected = true;
        printf("FIO detected. Switching to 10-second interval logging.\n");

        // Modify the timer's expiration and interval
        struct itimerspec its;
        its.it_value.tv_sec = 10; // Set next expiration to 10 seconds
        its.it_value.tv_nsec = 0;
        its.it_interval.tv_sec = 10; // Update interval to 10 seconds
        its.it_interval.tv_nsec = 0;

        if (timer_settime(timerid, 0, &its, NULL) == -1) {
            perror("timer_settime failed");
            exit(EXIT_FAILURE);
        }
        start_time = time(NULL); // Record start time for logging
        return;
    }

    // If FIO workload is detected, log statistics every 10 seconds
    if (fio_detected) {
        ++interval_count; // Increment the number of 10-second intervals processed

        // Record statistics in the buffer
        // Elapsed time since detecting FIO
        stats_buffer[stats_index].seconds = difftime(time(NULL), start_time);
        // Average I/O operations per second over 10 seconds
        stats_buffer[stats_index].iops = io_cnt / 10;
        // Write amplification factor over 10 seconds
        stats_buffer[stats_index++].waf = (double)(host_written_pages + gc_written_pages) / host_written_pages;

        // Reset counters for the next interval
        io_cnt = 0;
        host_written_pages = 0;
        gc_written_pages = 0;

        // Stop logging after 30 intervals (300 seconds)
        if (interval_count >= 30) {
            printf("FIO finished.\n");
            // print_write_lpn_access_cnt();
            print_statistics(); // Log the recorded statistics
            timer_delete(timerid); // Delete the timer to stop further logging
        }
    }
}
```

```c
void print_statistics(void) {
    // Open a file to log statistics(second, iops, waf)
    FILE *log_file = fopen("statistics.csv", "a");
    fprintf(log_file, "seconds,iops,waf\n");
    // Log the elapsed time, iops, waf in 10 second
    for (size_t i = 0; i < stats_index; i++) {
        fprintf(log_file, "%.0f,%ld,%.2f\n",
                stats_buffer[i].seconds,
                stats_buffer[i].iops,
                stats_buffer[i].waf);
    }
    fflush(log_file);
    fclose(log_file);
}
```

IOPS – Normal distribution 1.0



IOPS – Normal distribution 1.0 (except 0 second for clarity of graph trend)



IOPS Analysis: Normal distribution 1.0

All thresholds exhibit a sharp increase in IOPS within the first 10 seconds, stabilizing quickly after.

Threshold 0

Stabilizes between **18,000 and 18,200**.

Displays the least fluctuation among the thresholds.

Consistently the **lowest IOPS** across the thresholds.

Threshold 1

Fluctuates between **18,200 and 19,300**.

Exhibits higher fluctuations compared to Thresholds = 0 and 5.

Achieves the **highest peak IOPS**.

Threshold 5

Fluctuates between **18,000 and 19,000**.

Shows moderate fluctuation, with IOPS rising slightly in later intervals.

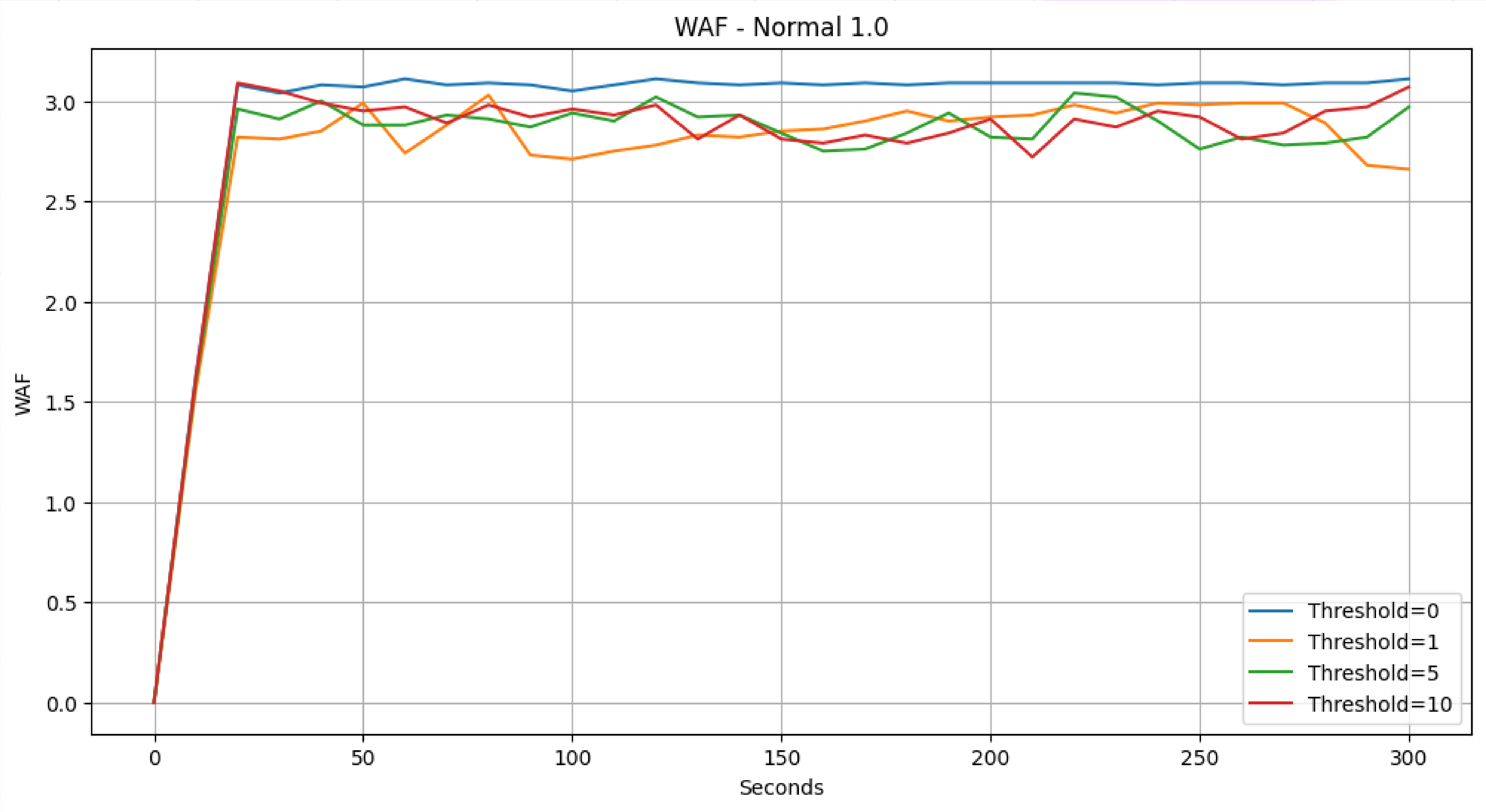
Threshold 10

Fluctuates within the range of **18,000 to 19,000**.

Comparable to Threshold = 5, slightly lower on average.

Does not surpass Threshold = 1 and 5 in most cases.

WAF – Normal distribution 1.0



WAF Analysis: Normal distribution 1.0

Threshold 0

Stabilizes between **3.0 and 3.1.**

Shows minimal fluctuation.

Consistently exhibits the **highest WAF**, indicating the least efficient write amplification handling.

Threshold 1

Fluctuates between **2.7 and 3.0.**

Demonstrates improved efficiency compared to Threshold = 0, with a lower WAF throughout.

Threshold 5

Stabilizes between **2.75 and 3.05**.

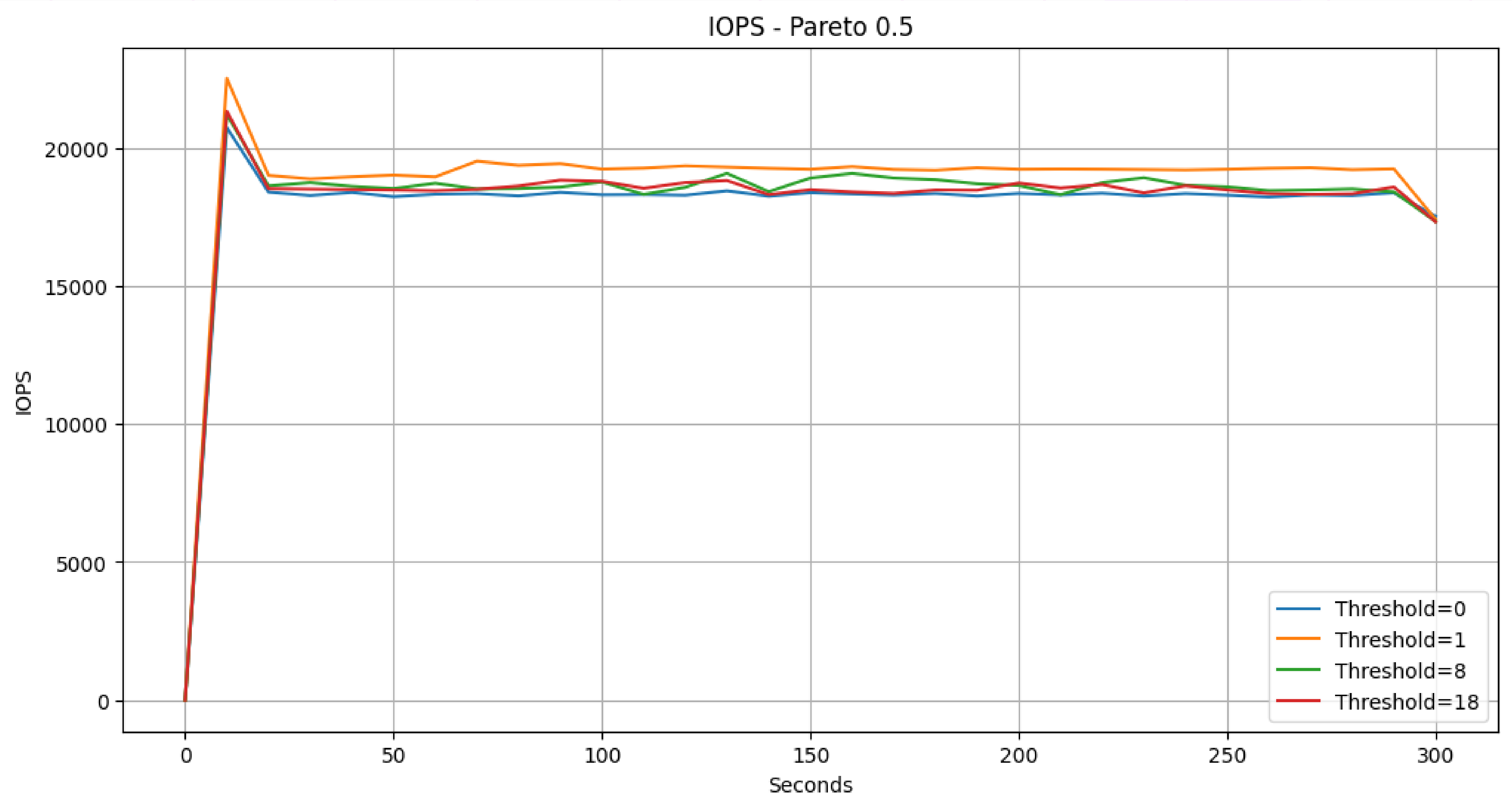
Shows slight fluctuations but remains relatively stable, similar to Threshold = 10.

Threshold 10

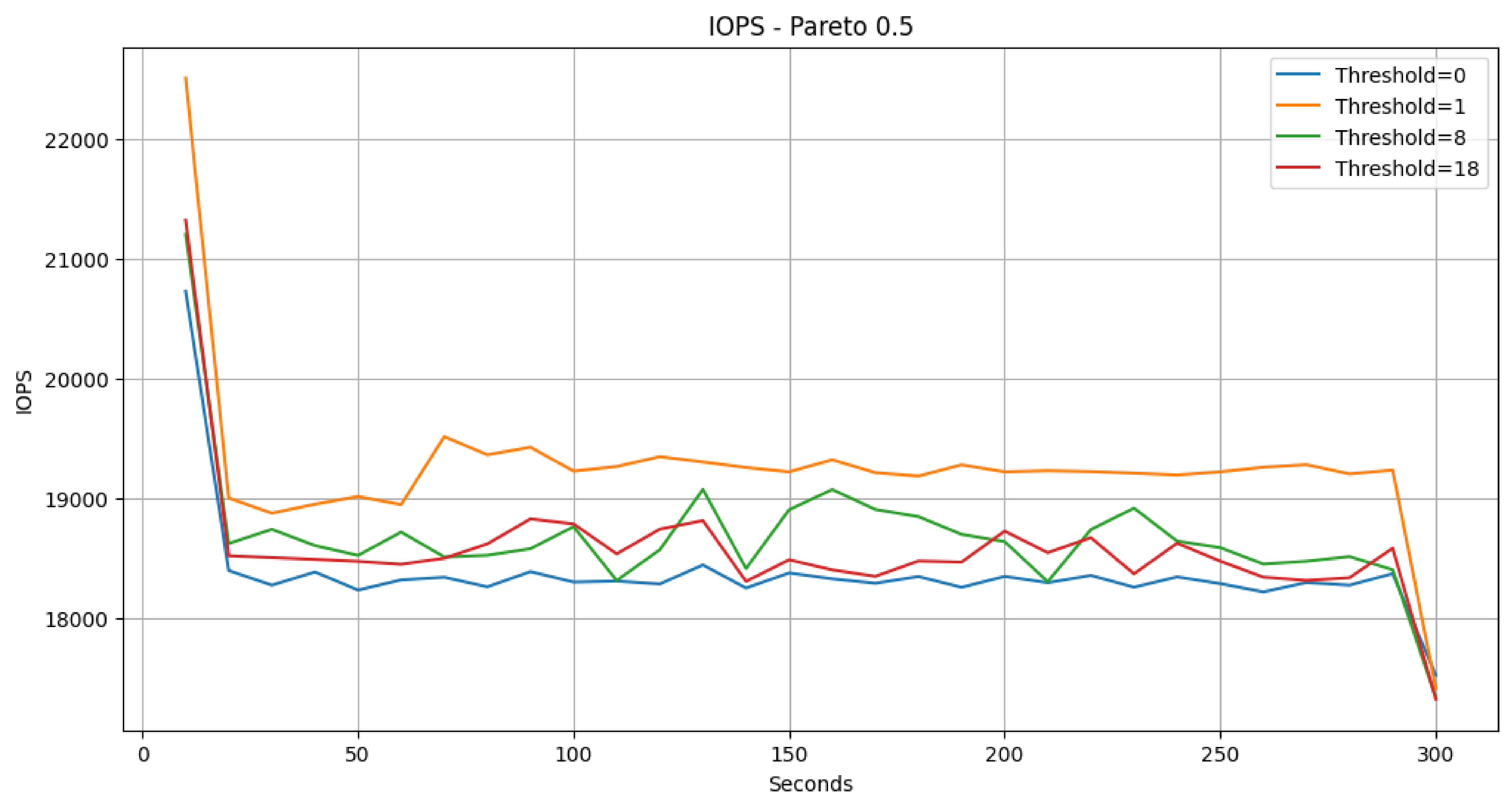
Stabilizes between **2.7 and 3.1**.

Performs similarly to Threshold = 5 in terms of efficiency and stability.

IOPS – Pareto distribution 0.5



IOPS – Pareto distribution 0.5 (0-second data excluded for clarity of graph trends)



IOPS Analysis: Pareto distribution 0.5

All thresholds exhibit a sharp increase in IOPS within the first 10 seconds, stabilizing quickly after.

Threshold 0

Fluctuates between **18,000 and 18,500** IOPS.

Provides consistent performance but exhibits **the lowest IOPS** compared to other thresholds.

Maintains a relatively stable trend with minimal variation.

Threshold 1

Fluctuates between **18,500 and 19,000,** peaking near **19,500.**

Consistently achieves **the highest IOPS**.

Threshold 8

Fluctuates between **18,300 and 19,000** IOPS.

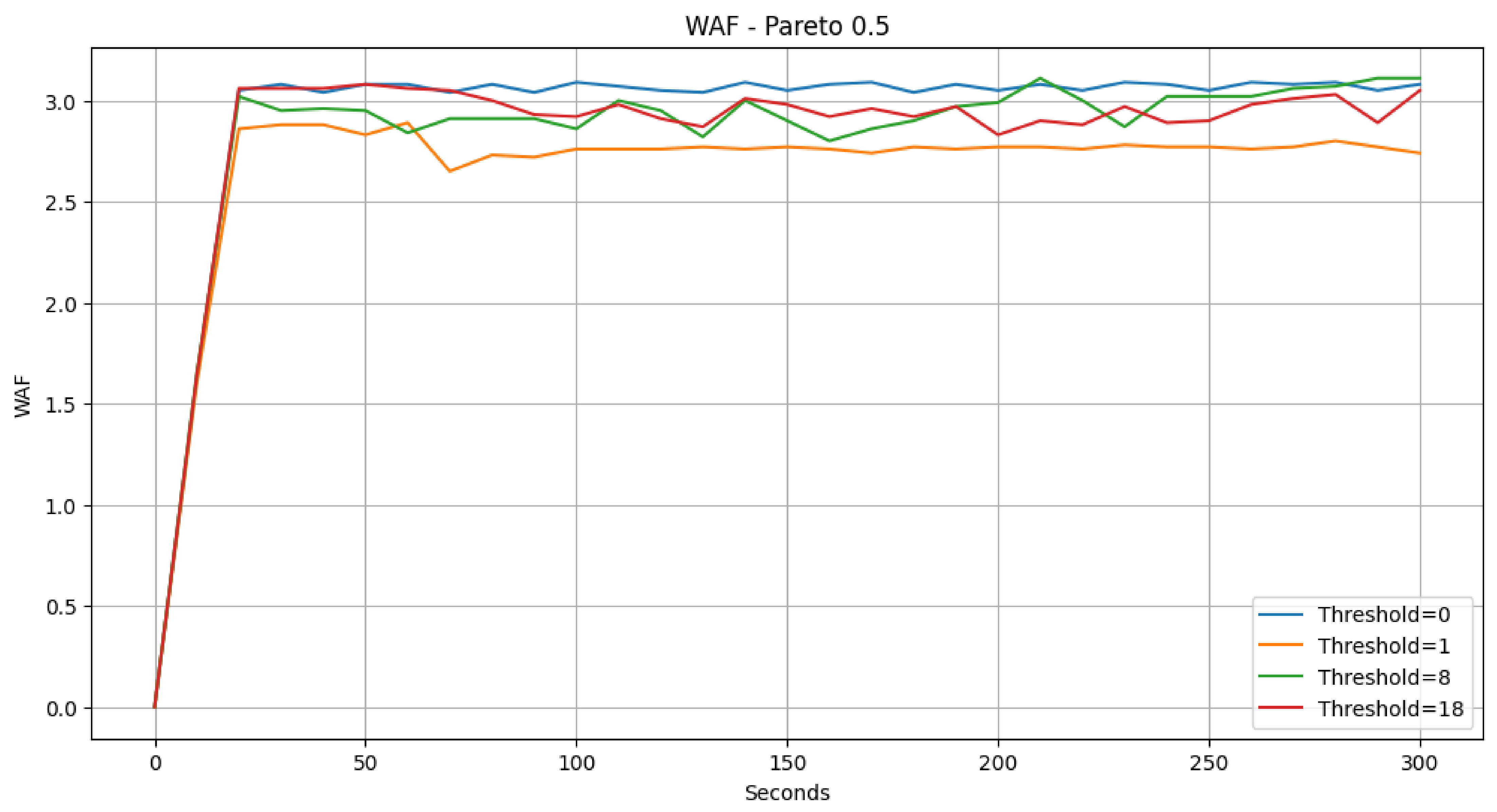
Demonstrates moderate performance with occasional spikes and dips

Threshold 18

Fluctuates between **18,300 and 18,900** IOPS.

Similar performance to Threshold 8, but with slightly more pronounced fluctuations.

WAF – Pareto distribution 0.5



WAF Analysis: Pareto distribution 0.5

Threshold 0

Stabilizes between **3.0 and 3.1.**

Consistently exhibiting the **highest WAF,** indicating **the least** efficient write amplification handling.

Stability is high.

Threshold 1

Stabilizes between **2.7 and 2.9.**

Achieves the **lowest WAF** across all thresholds, demonstrating the **best** write efficiency.

Threshold 8

Fluctuates between **2.8 and 3.1,** with occasional spikes beyond **3.0**.

Higher variability compared to other thresholds, with noticeable spikes at intervals.

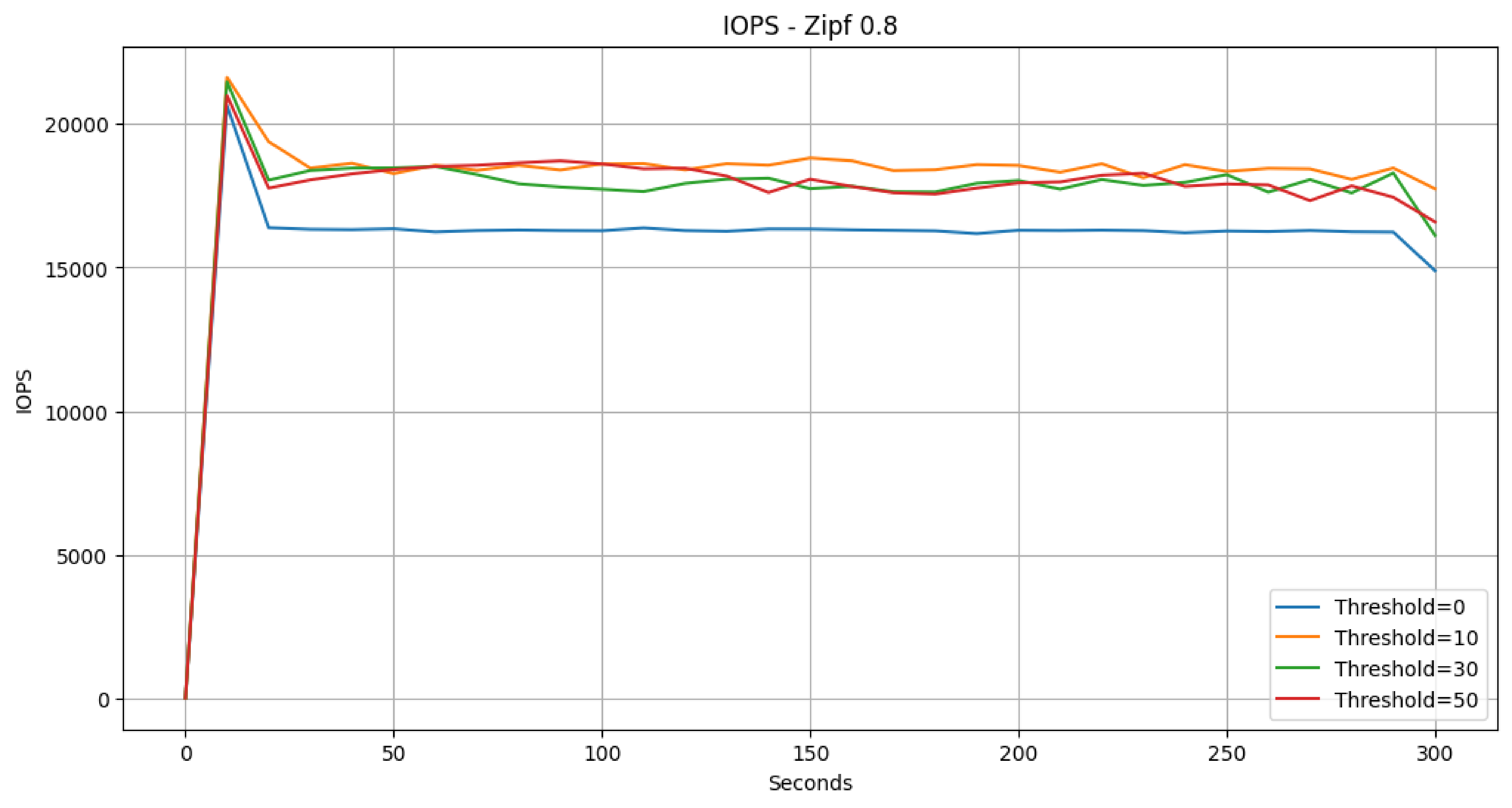
Threshold 18

Fluctuates between **2.8 and 3.0**,

Closely aligning with Threshold 8 but with slightly more consistent behavior.

Offers similar efficiency to Threshold 8, but less pronounced spikes, resulting in slightly better stability.

IOPS – Zipf Distribution 0.8



IOPS Analysis: Zipf distribution 0.8

All thresholds exhibit a sharp increase in IOPS within the first 10 seconds, stabilizing quickly after.

Threshold 0

Stabilizes between **15,000 and 16,000** IOPS.

Provides consistent performance but exhibits **the lowest IOPS** compared to other thresholds.

Threshold 10

Stabilizes between **18,000 and 19,400** IOPS**.** Achieves the **highest IOPS** across all thresholds.

Maintains a relatively stable trend with minimal variation.

Threshold 30

Stabilizes between **17,500 and 18,500** IOPS.

Exhibits slightly lower IOPS than Threshold 10 but maintains good stability.

Threshold 50

Stabilizes between **17,300 and 18,700** IOPS.

Similar performance to Threshold 30, with slightly better throughput but more variability.

-> The corresponding IOPS graph shows that the hot-cold separation effect of Zipf 0.8 is excellent, with a significant number of accesses concentrated on a specific LPN.

WAF – Zipf Distribution 0.8

WAF Analysis: Zipf distribution 0.8

Threshold 0

Stabilizes between **3.7 and 3.8.**

Consistently exhibits the **highest WAF**, indicating **the least efficient** write amplification handling among all thresholds.

Threshold 10

Stabilizes between **2.6 and 3.2.**

Slightly less efficient than Thresholds 30 and 50 but provides moderate stability across intervals.

Threshold 30

Stabilizes between **2.6 and 3.0.**

Achieves better efficiency compared to Thresholds 10.

Threshold 50

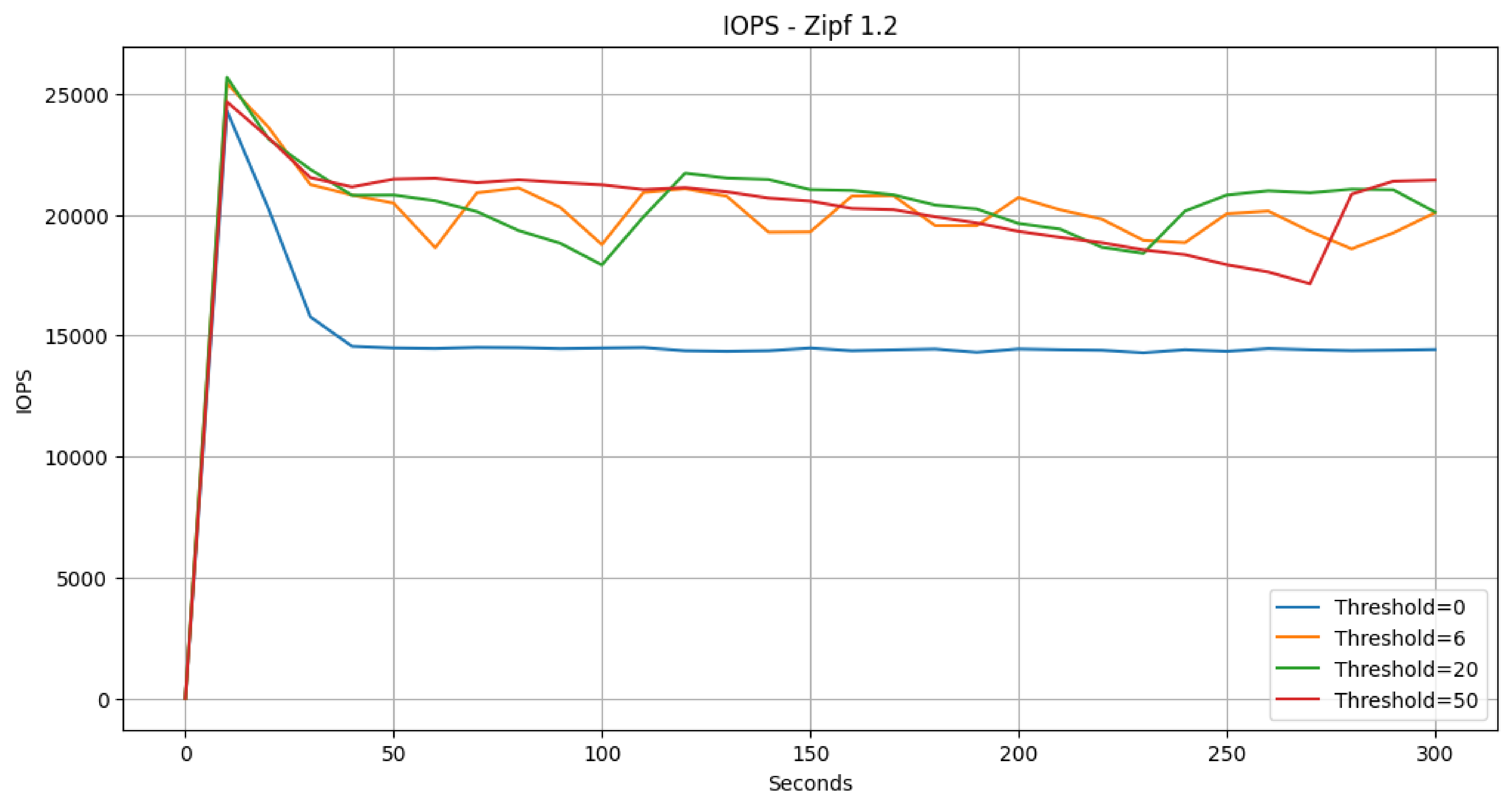
Stabilizes between **2.6 and 3.1.**

Similar to Threshold 30 in efficiency but exhibits slightly more fluctuations in the mid-to-late intervals.

Achieves the **lowest WAF** overall, indicating the highest write efficiency among all thresholds.

-> The corresponding WAF graph shows that the hot-cold separation effect of Zipf 0.8 is excellent, with a significant number of accesses concentrated on a specific LPN.

IOPS – Zipf Distribution 1.2



IOPS Analysis: Zipf distribution 1.2

All thresholds exhibit a sharp increase in IOPS within the first 10 seconds, stabilizing quickly after.

Threshold 0

Stabilizes around **15,000** IOPS**.**

Exhibits the **lowest IOPS** among all thresholds.

Threshold 6

Stabilizes between **19,000 and 21,000** IOPS**.**

Offers a moderate performance compared to higher thresholds.

Threshold 20

Fluctuates between **19,000 and 21,000** IOPS, showing moderate variability.

Exhibits slightly higher variability compared to Threshold 6 but maintains high average IOPS.

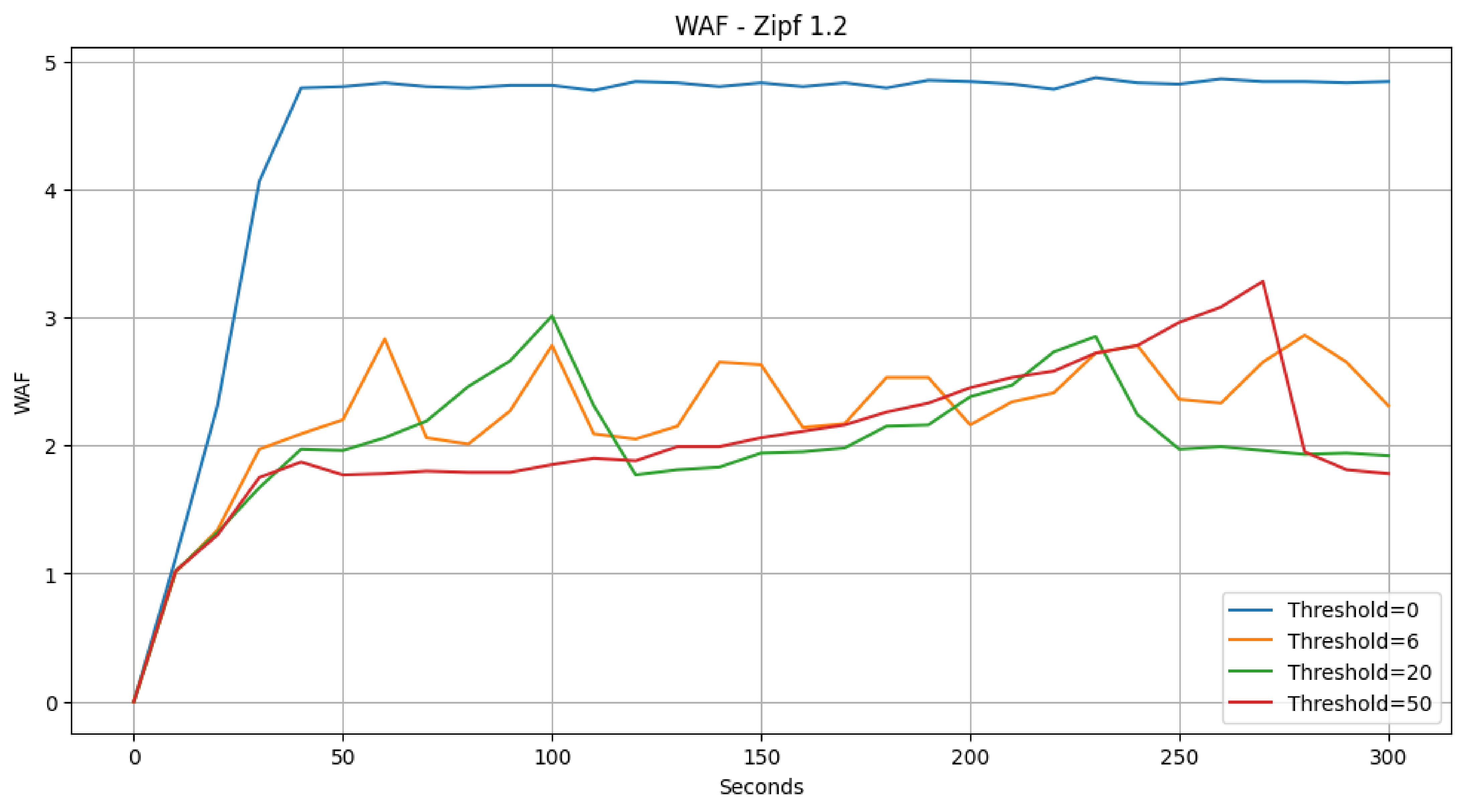
Threshold 50

Fluctuates between **19,500 and 21,000** IOPS, with noticeable peaks and dips.

Shows a pattern of IOPS variation in a longer cycle compared to other thresholds.

-> The corresponding IOPS graph shows that the hot-cold separation effect in Zipf 1.2, where a significant number of accesses are concentrated on a particular LPN, is remarkably superior compared to other distributions.

WAF – Zipf Distribution 1.2



WAF Analysis: Zipf distribution 1.2

Threshold 0

Stabilizes around **4.8 to 5.0** after the initial phase.

Exhibits the **highest WAF** among all thresholds, indicating the **least efficient** write amplification handling among all thresholds.

Threshold 6

Fluctuates between **2.0 and 2.9**, with moderate variability.

It shows a pattern of WAF variation in a shorter cycle compared to other thresholds.

Threshold 20

Fluctuates between **1.9 and 3.0**, with noticeable peaks and dips.

Balances relatively low WAF with moderate variability.

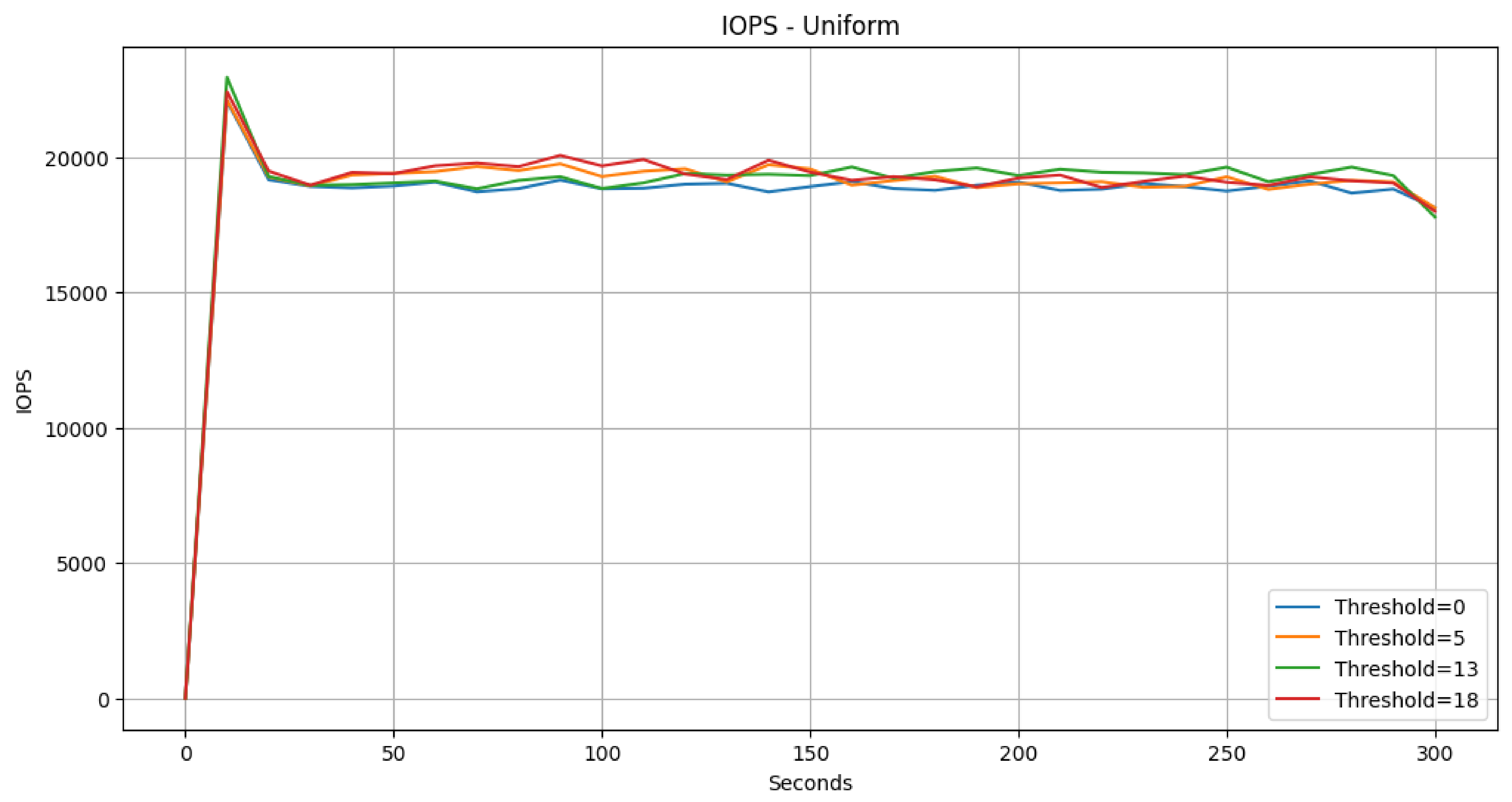
Threshold 50

Fluctuates widely between **1.9 and 3.2**, with significant spikes.

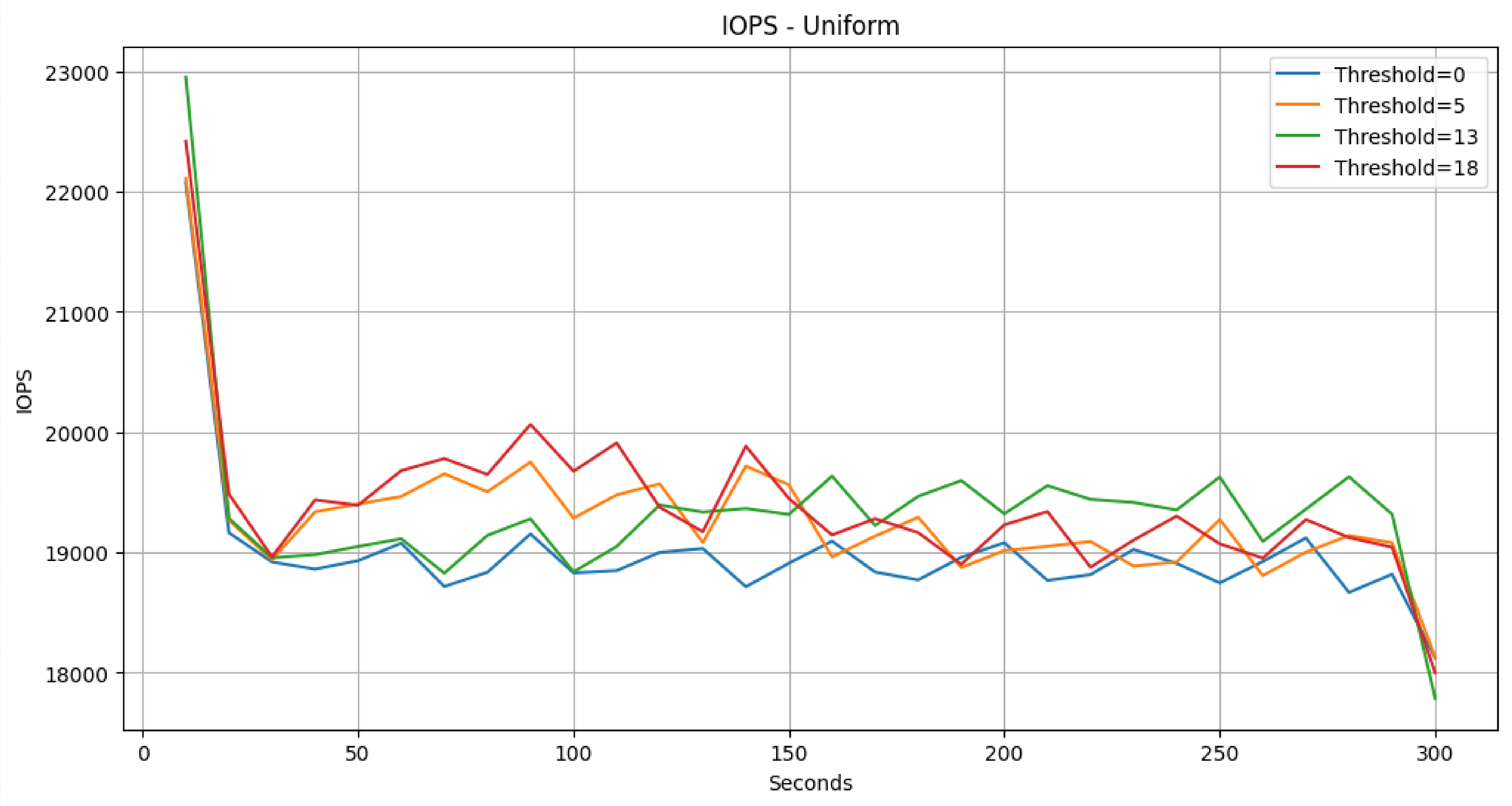
It shows a pattern of WAF variation in a longer cycle compared to other thresholds.

-> The corresponding WAF graph shows that the hot-cold separation effect in Zipf 1.2, where a significant number of accesses are concentrated on a particular LPN, is remarkably superior compared to other distributions.

IOPS – Uniform distribution



IOPS – Uniform distribution (0-second data excluded for clarity of graph trends)



IOPS Analysis: Uniform distribution

All thresholds exhibit a sharp increase in IOPS within the first 10 seconds, stabilizing quickly after.

Threshold 0

Fluctuates between **18,600 and 19,100**.

Consistently delivers the **lowest IOPS** across all thresholds.

Threshold 5

Fluctuates between **18,900 and 19,800**.

Balances performance and stability but does not achieve the peak efficiency of higher thresholds.

Threshold 13

Fluctuates between **18,900 and 19,600.**

Maintains a balance between high throughput and moderate variability.

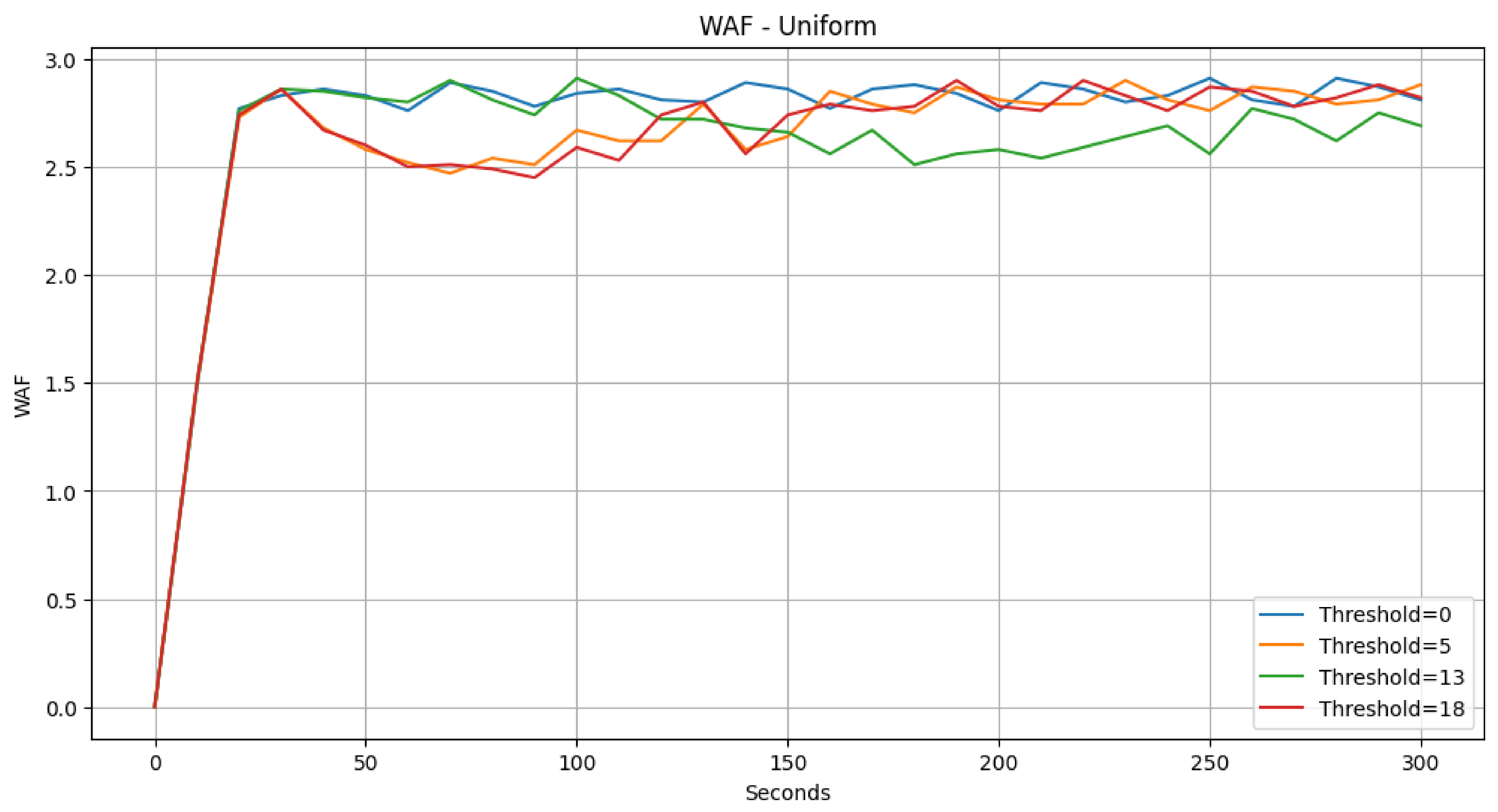
Threshold 18

Fluctuates between **18,800 and 20,000** with occasional dips and peaks.

Performance closely resembles Threshold 13 but with slightly higher variability.

-> The IOPS graph indicates that the impact of hot-cold separation is relatively small in the uniform distribution, as it features a similar number of accesses across LPNs compared to other distributions.

WAF – Uniform distribution



WAF Analysis: Uniform distribution

Threshold 0

Stabilizes between **2.76 and 2.9** after the initial phase.

Consistently exhibits the **highest WAF** across all thresholds.

Threshold 5

Fluctuates between **2.47 and 2.9.**

Moderate stability with slight fluctuations. WAF values sometimes overlap Threshold 0 in certain sections.

Threshold 13

Fluctuates between **2.51 and 2.9**.

Show balanced performance in both WAF and stability.

Threshold 18

Fluctuates between **2.45 and 2.9.**

WAF values sometimes overlap Threshold 0 in certain sections.

-> The WAF graph indicates that the impact of hot-cold separation is relatively small in the uniform distribution, as it features a similar number of accesses across LPNs compared to other distributions.