# Heterogeneous(이종) 프로세서 환경에서 WAL 동기화 정책이 성능에 미치는 영향

**RocksDB Study 희로애락**

**윤치호 박규원 강민수 이우진**

2026. 01. 21

DANKOOK UNIVERSITY

Dankook University System Software Laboratory

# Contents

# Why WAL? — Durability(내구성) 요구

Main memory is volatile → 만약 전원 꺼짐 혹은 문제가 발생한다면?
**Lost user data is a big risk in a DB company !!**

Require durability : 어떤 데이터가 쓰여질 때, 전원 결함 같은 문제가 발생하더라도 데이터를 다시 이용할 수 있도록 보장해 주는 것

**Data durability를 보장하는 RocksDB의 방법 =**
**WAL (Write-Ahead-Log)**

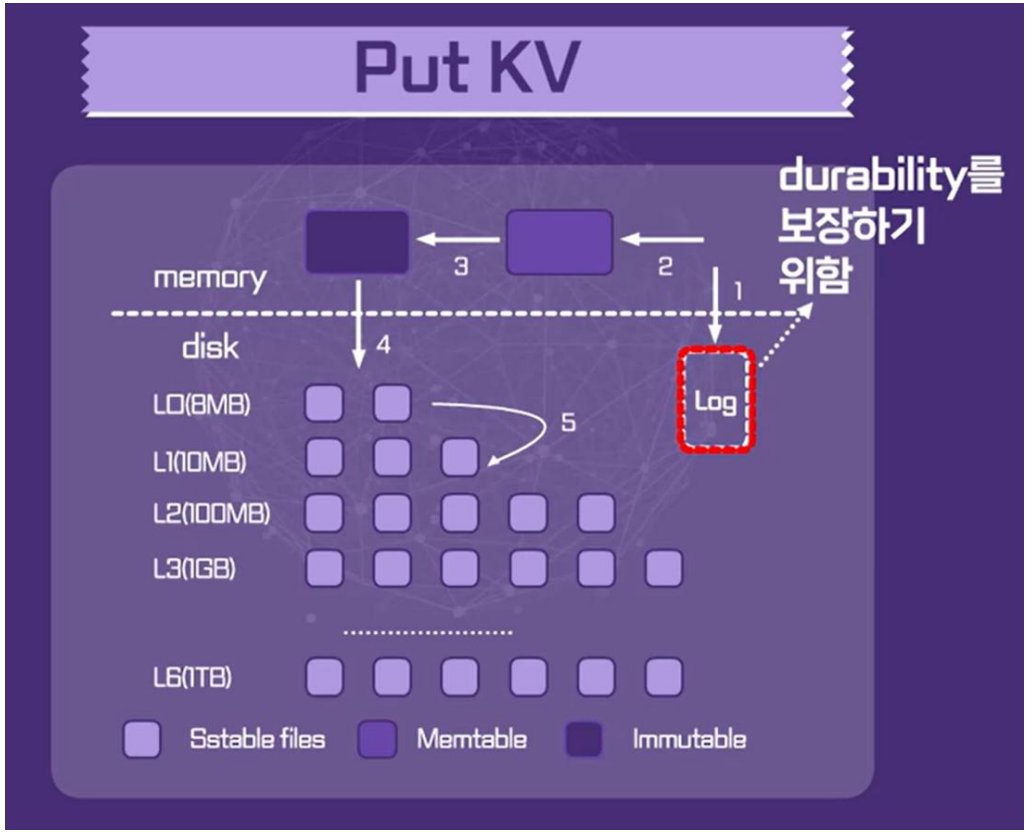
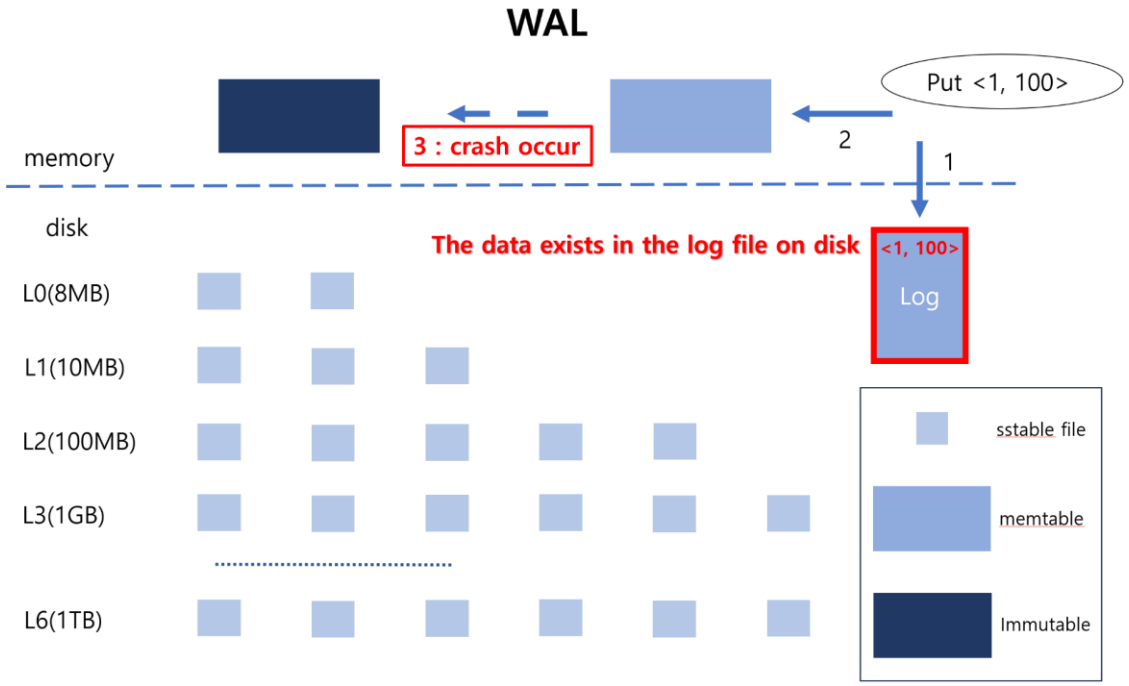



*그림: Put KV 경로에서 WAL(Log)의 역할 (user provided)*

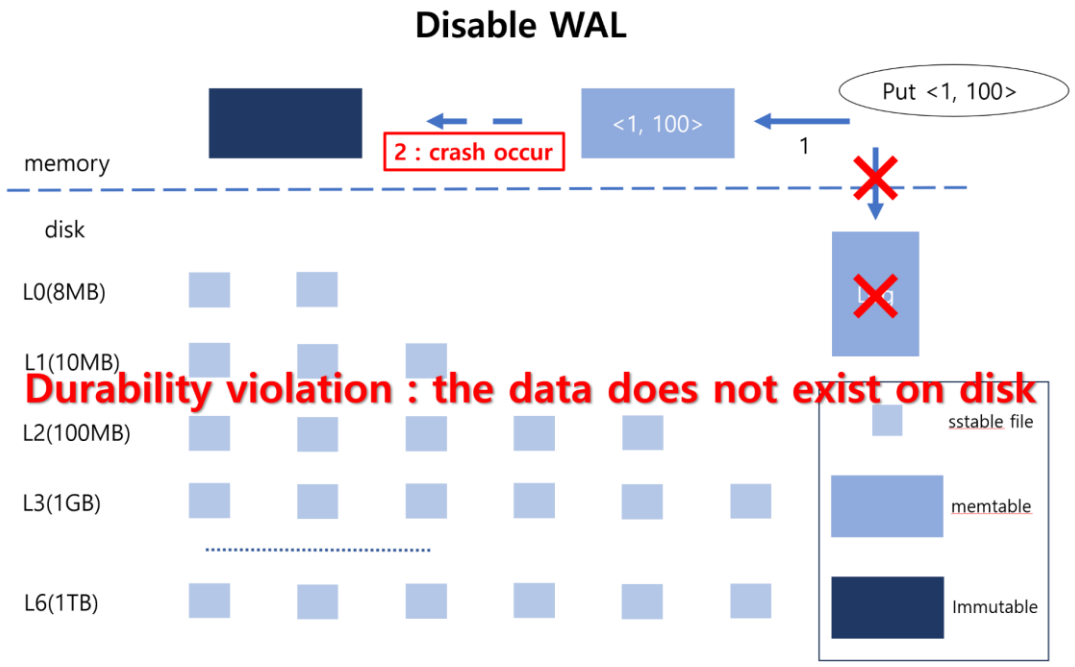
# WAL의 역할 — Write path와 Crash 복구

- Write는 WAL에 먼저 append → memtable 반영
- memtable이 SST로 안전하게 flush되면, 그 memtable에 대응하는 WAL은 obsolete가 되어 아카이브되고, 일정 기간 후 삭제
- Crash 발생 시 WAL을 repla하여 memtable 재구성
- “디스크에 남은 로그”가 복구 근거



# Disable WAL 시 위험 — Durability violation

- DisableWAL=true: WAL 기록 자체를 생략
- 크래시/전원 장애가 flush 전에 발생하면 복구 데이터가 없음
- 성능은 좋아질 수 있으나, 내구성 요구가 있는 서비스에는 부적합



# WAL 파일 구조 — Record / Block


- WAL file은 가변 길이 record들의 연속
- Record format: CRC(4B) · Size(2B) · Type(1B) · Payload- CRC = payload(type+data에 대한 crc32c) 기반 체크섬
- - Size = payload의 길이
- - Type = record 조각 타입(FULL/FIRSRT/MIDDLE/LAST 등)
- - Payload = <key, value> 쌍이 들어가는 영역

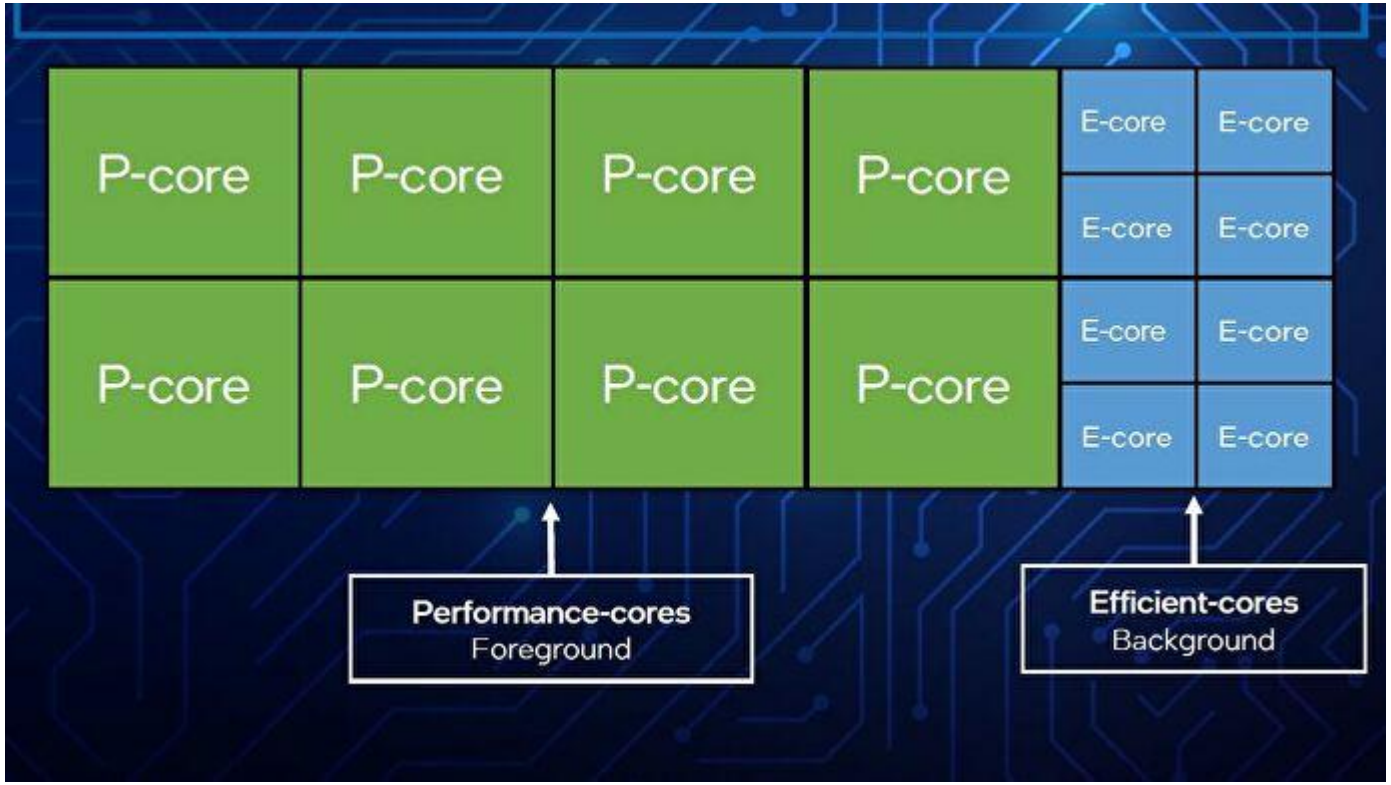
*Record header format (user provided)*

# WAL 병목 지점 (왜 sync=true가 느린가?)

| 구분 | 항목 | 원인/메커니즘 | 증상(관측) |
|---|---|---|---|
| **병목** | **Sync 모드 = fsync(fdatasync) 대기** | `sync=true`이면 WAL append 후 **fsync 완료까지 blocking**. Group commit이 있어도 **fsync 비용 자체**는 남음 | ops/sec 급락, 평균/꼬리 지연(p99) 급증 |
| | Writer Queue 대기(리더-팔로워) | 동시 write 시 **leader 1개가 WAL+memtable 처리**, 나머지는 leader 완료까지 대기 | 스레드/코어 늘려도 처리량이 잘 안 늘고 대기 시간 누적 |
| | WAL CPU 오버헤드(CRC32C + 헤더) | WAL fragment마다 **CRC32C 계산 + 헤더 처리**. 작은 write가 많을수록 호출 횟수↑ | CPU 사용 증가, small write에서 오버헤드가 두드러짐 |
| | flush/동기화 호출 빈도(버퍼 경로) | 작은 write가 매우 많으면 **flush/동기화 관련 호출이 빈번**해져 병목 강화(특히 sync 모드) | latency 분산↑, p99/p99.9 상승 |
| | WAL I/O 경합(디스크 경쟁) | WAL append와 **SST flush/compaction write**가 같은 디스크에서 큐 경쟁 | compaction/flush가 겹칠 때 지연이 갑자기 커짐 |

**핵심: sync=true는 I/O bound → 코어 차이가 가려지고, sync=false는 CPU/스케줄링 영향이 커짐**

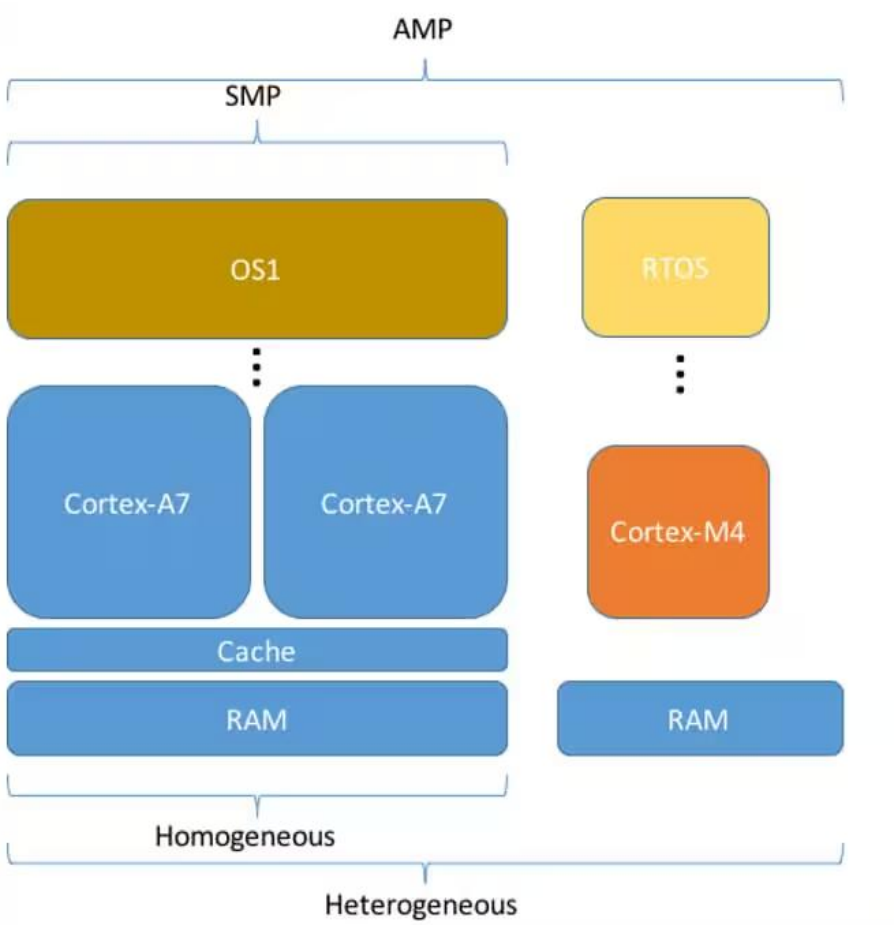
# What is P-core & E-core ?

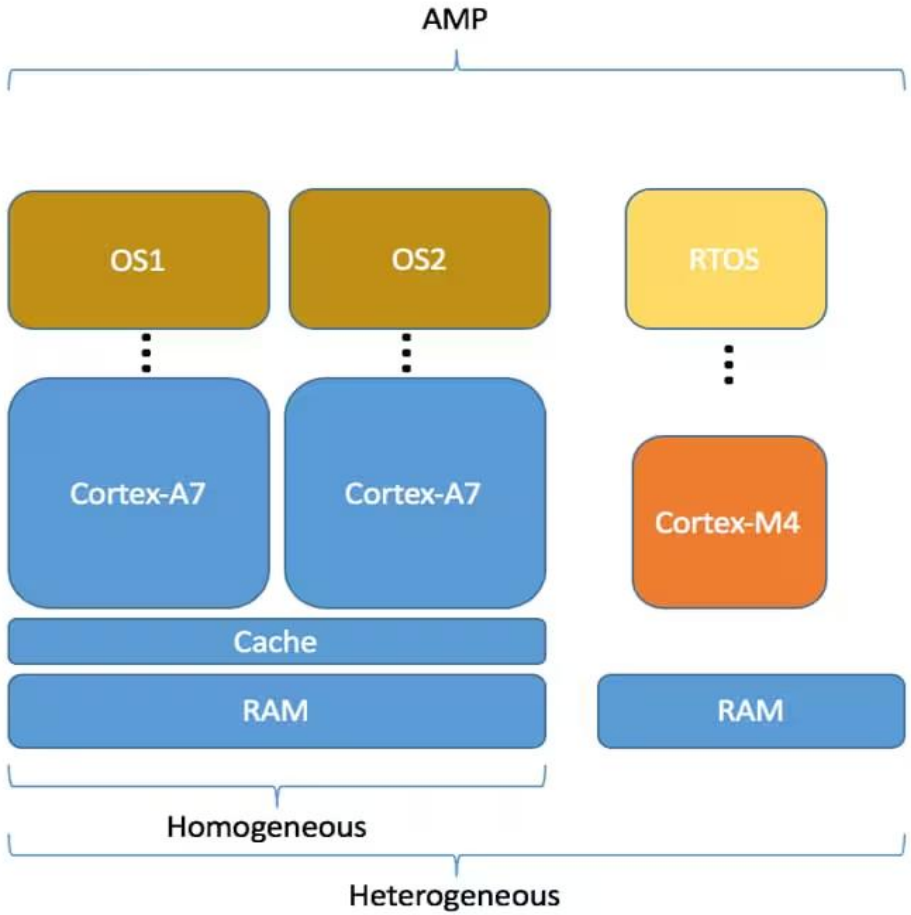



- **P-core(Performance-core)**
  고성능/고클럭·고IPC 중심으로 설계된 코어
  무거운 단일 스레드 작업(게임 엔진, 반응성 작업)에 유리
  하이퍼스레딩 지원 가능

- **E-core(Efficient-core)**
  전력 효율 중심의 코어 (P-core 공간에 여러 개 탑재 가능)
  멀티 스레드 확장/병렬 작업과 백그라운드 작업 처리에 유리
  가벼운 작업을 맡아 P-core를 중요한 작업에 집중시키는 역할

# 이기종 프로세서 환경(Heterogeneous Computing)

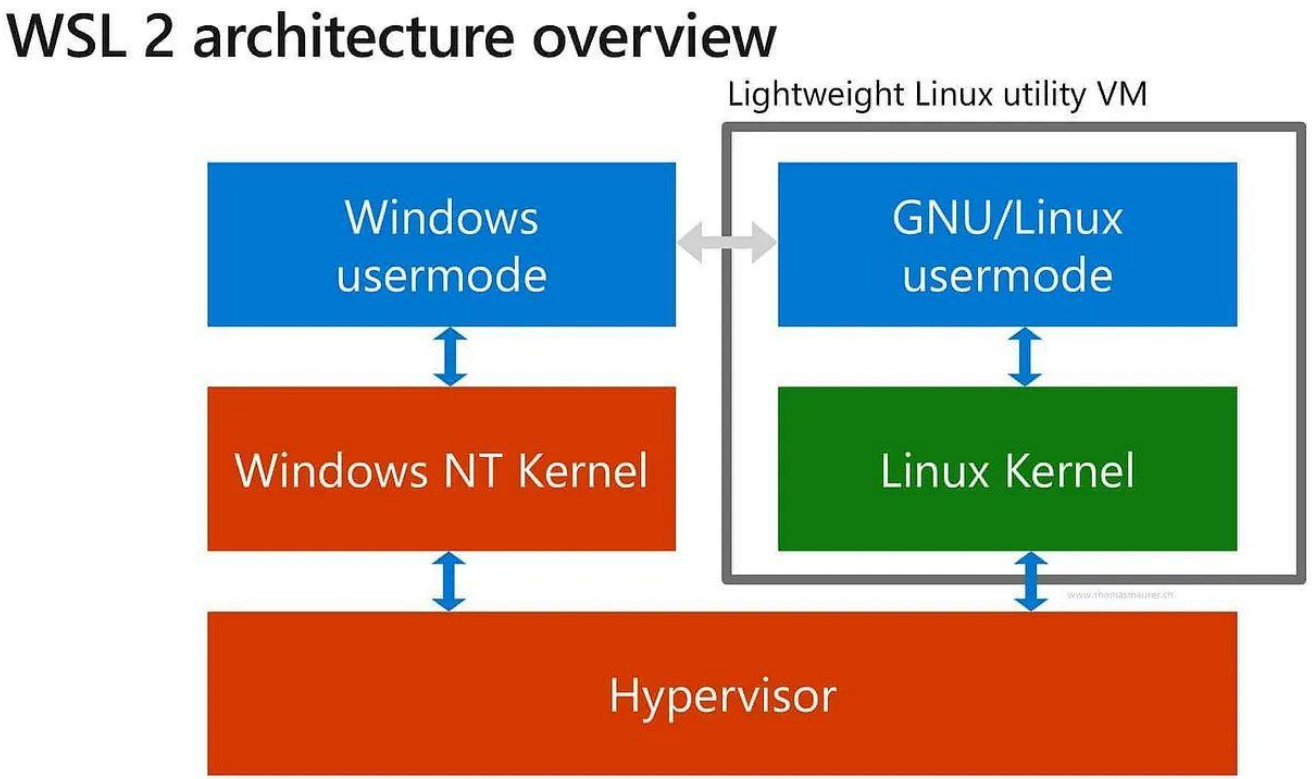


# 가상화


VM1
애플리케이션 1
바이너리/
라이브러리
게스트
운영체제
VM2
애플리케이션 1
바이너리/
라이브러리
게스트
운영체제
VM3
애플리케이션 1
바이너리/
라이브러리
게스트
운영체제
하이퍼바이저
호스트 운영체제
하드웨어
호스트 가상화
VM1
애플리케이션 1
바이너리/
라이브러리
게스트
운영체제
VM2
애플리케이션 1
바이너리/
라이브러리
게스트
운영체제
VM3
애플리케이션 1
바이너리/
라이브러리
게스트
운영체제
하이퍼바이저
하드웨어
하이퍼바이저 가상화
컨테이너 1
애플리케이션 1
바이너리/
라이브러리
컨테이너 2
애플리케이션 1
바이너리/
라이브러리
컨테이너 3
애플리케이션 1
바이너리/
라이브러리
도커
운영체제
하드웨어
컨테이너 가상화

# 가상화


WSL 2 architecture overview
Lightweight Linux utility VM
Windows usermode
GNU/Linux usermode
Windows NT Kernel
Linux Kernel
Hypervisor

# Experimental configuration

- **asus vivobook s16 oled**
- **16gb ram**
- **ultra 7 155h**
- **1tb ssd MTFDKBA1T0QFM-1BD1AABYY pcie4 qlc windows 11**
- **wsl2 . Ubuntu 24.04**
- **db_bench tool**
  Fillrandom,overwrite

CPU 사양

| | |
|---|---|
| 코어 수 | 16 |
| Performance-core의 수 | 6 |
| Efficient-core의 수 | 8 |
| 저전력 Efficient-cores 수 | 2 |
| 총 스레드 수 | 22 |
| 최대 터보 주파수 | 4.8 GHz |
| Performance-core 최대 터보 주파수 | 4.8 GHz |
| Efficient-core 최대 터보 주파수 | 3.8 GHz |
| 저전력 Efficient-core 최대 터보 주파수 | 2.5 GHz |
| Performance-core 기본 주파수 | 1.4 GHz |
| Efficient-core 기본 주파수 | 900 MHz |
| 저전력 Efficient-core 기본 주파수 | 700 MHz |

# Experimental control

.wslconfig

C: > Users > ych97 > .wslconfig

```
1    [ws12]
2    processors=4
```

프로세서 선호도

"vmmemWSL"을(를) 실행할 수 있는 프로세서는 무엇인가요?

- [ ] 모든 프로세서
- [x] CPU 0
- [x] CPU 1
- [x] CPU 2
- [x] CPU 3
- [ ] CPU 4

확인 취소

CPU 사양

| | |
|---|---|
| 코어 수 | 16 |
| Performance-core의 수 | 6 |
| Efficient-core의 수 | 8 |
| 저전력 Efficient-cores 수 | 2 |
| 총 스레드 수 | 22 |

| CPU [#0]: Intel Core Ultra 7 155H | | | | |
|---|---|---|---|---|
| Core VIDs | 0.773 V | 0.580 V | 1.120 V | 0.809 V |
| Uncore VID | 0.923 V | 0.871 V | 0.926 V | 0.922 V |
| 코어 클럭 | 2,313.6 MHz | 399.0 MHz | 4,787.8 MHz | 2,519.9 MHz |
| E-core 0 클럭 | 1,796.0 MHz | 997.6 MHz | 3,790.7 MHz | 1,880.6 MHz |
| E-core 1 클럭 | 1,796.0 MHz | 1,496.3 MHz | 3,790.9 MHz | 1,976.7 MHz |
| E-core 2 클럭 | 1,796.0 MHz | 1,496.1 MHz | 3,691.0 MHz | 2,009.9 MHz |
| E-core 3 클럭 | 1,796.0 MHz | 1,496.1 MHz | 3,790.7 MHz | 1,967.4 MHz |
| E-core 4 클럭 | 1,496.7 MHz | 997.5 MHz | 3,790.8 MHz | 1,917.6 MHz |
| E-core 5 클럭 | 1,496.7 MHz | 1,496.3 MHz | 3,790.8 MHz | 2,048.7 MHz |
| E-core 6 클럭 | 1,796.0 MHz | 1,496.4 MHz | 3,791.0 MHz | 2,061.7 MHz |
| E-core 7 클럭 | 1,796.0 MHz | 1,496.4 MHz | 3,791.0 MHz | 2,078.3 MHz |
| P-core 8 클럭 | 4,589.8 MHz | 1,795.6 MHz | 4,787.8 MHz | 4,376.4 MHz |
| P-core 9 클럭 | 4,589.8 MHz | 1,696.0 MHz | 4,787.8 MHz | 4,424.4 MHz |
| P-core 10 클럭 | 4,490.1 MHz | 1,496.4 MHz | 4,490.1 MHz | 3,944.1 MHz |
| P-core 11 클럭 | 1,995.6 MHz | 1,496.4 MHz | 4,289.9 MHz | 3,365.9 MHz |
| P-core 12 클럭 | 1,796.0 MHz | 1,496.4 MHz | 4,289.8 MHz | 2,894.8 MHz |
| P-core 13 클럭 | 3,691.8 MHz | 1,396.7 MHz | 4,290.2 MHz | 3,417.6 MHz |
| E-core (LP) 14 클럭 | 997.8 MHz | 498.8 MHz | 1,097.5 MHz | 962.5 MHz |
| E-core (LP) 15 클럭 | 1,097.6 MHz | 399.0 MHz | 1,097.6 MHz | 992.0 MHz |

# Experiment shell script Part1

```
15 NUM_FIXED_THREADS=4
14
13 NUM_ENTRIES=50000
12 DB_BASE_PATH="/tmp/rocksdb_test_quick_4core"
11 LOG_DIR="logs_quick_4core"
10
 9 # 공통 db_bench 옵션
 8 COMMON_OPTIONS="--key_size=16 \
 7 --value_size=100 \
 6 --write_buffer_size=268435456 \
 5 --cache_size=1073741824 \
 4 --max_background_jobs=8 \
 3 --compression_type=snappy \
 2 --statistics \
 1 --histogram \
21   --stats_interval_seconds=5"
```

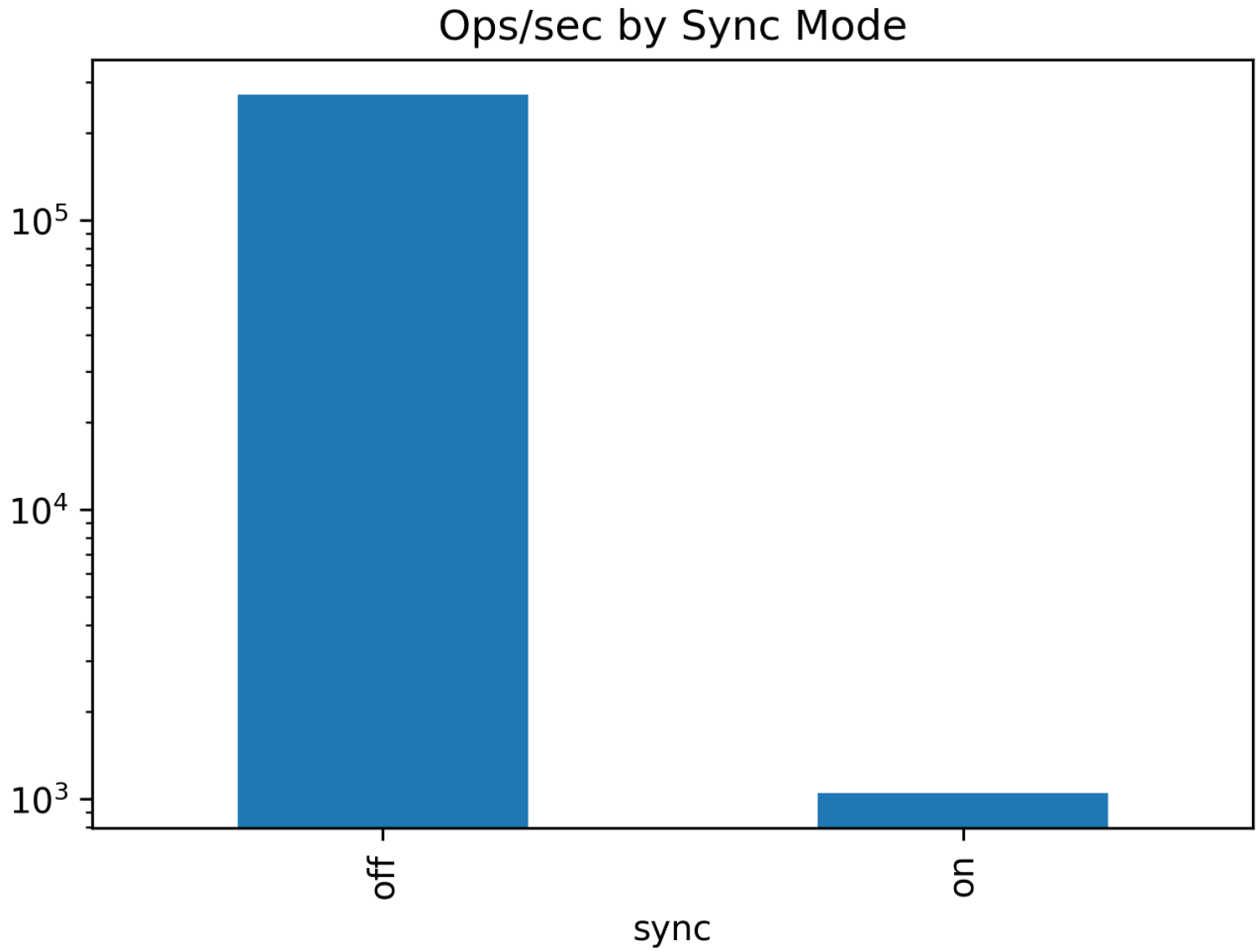
# Experiment shell script Part2

```
scenarios=(
    "sync_mode;true;$NUM_FIXED_THREADS"
    "async_mode;false;$NUM_FIXED_THREADS"
)

for scenario_info in "${scenarios[@]}"; do
    IFS=';' read -r name sync_mode threads <<< "$scenario_info"

    DB_PATH="${DB_BASE_PATH}_${name}"
    LOG_FILE="${LOG_DIR}/scenario_${name}.log"

    echo "-------------------------------------------------------"
    echo "Running Scenario: [${name}] with ${threads} threads"

    rm -rf "$DB_PATH"

    COMMAND="./db_bench \
      --benchmarks=fillrandom,overwrite \
      --num=$NUM_ENTRIES \
      --threads=$threads \
      --db=$DB_PATH \
      --sync=$sync_mode \
      $COMMON_OPTIONS"

    echo "Executing: $COMMAND"

    eval "$COMMAND" > "$LOG_FILE" 2>&1

    echo "Scenario [${name}] finished."
    sleep 2
done
```

# Hypotheses

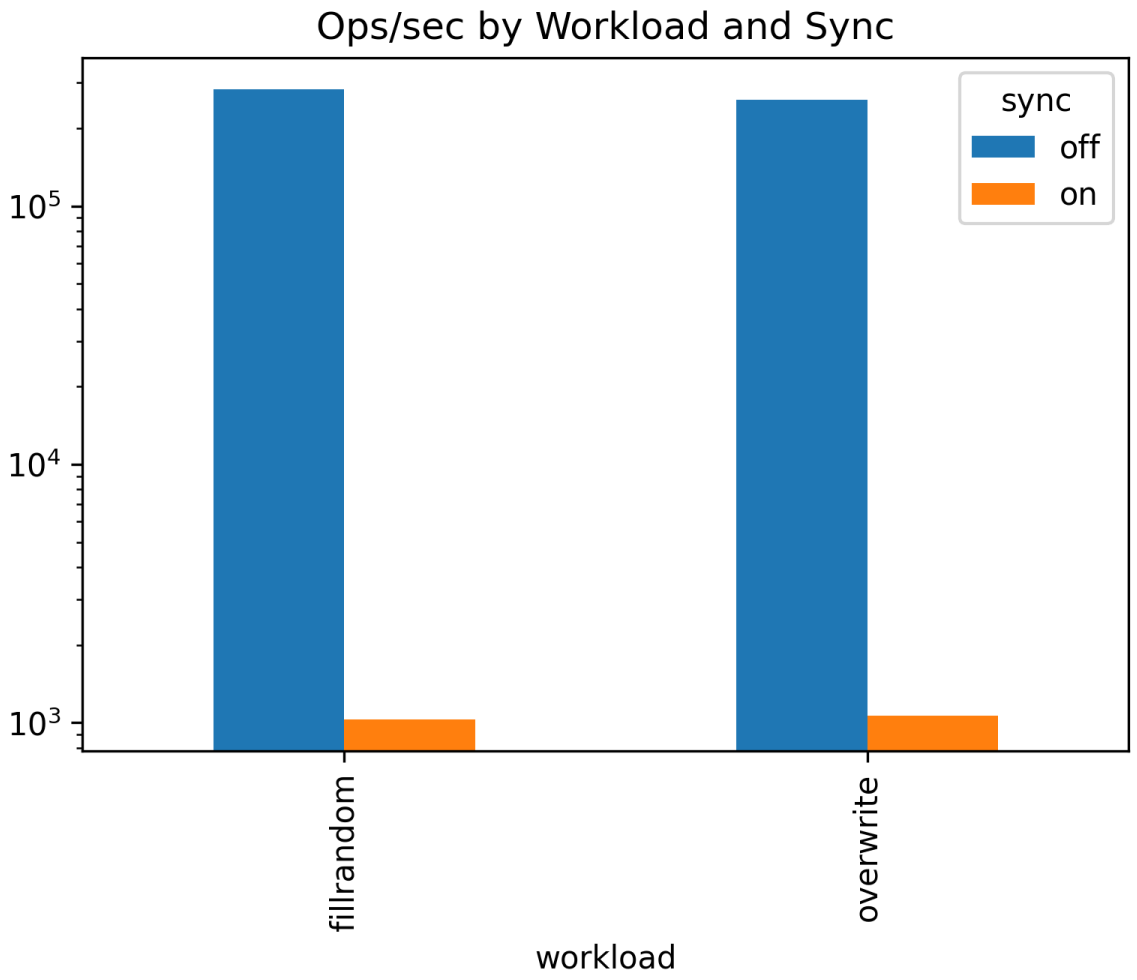
- Sync(on) & Async(off)의 병목에 따른 속도 차이가 있을 것이다
- Sync(on)에서는 디스크 병목이 지배 → 코어 차이가 작다
- 성능 순위 예상: P > (설정 없음) > E

# 결과 1 - Sync 모드별 Throughput(ops/sec)

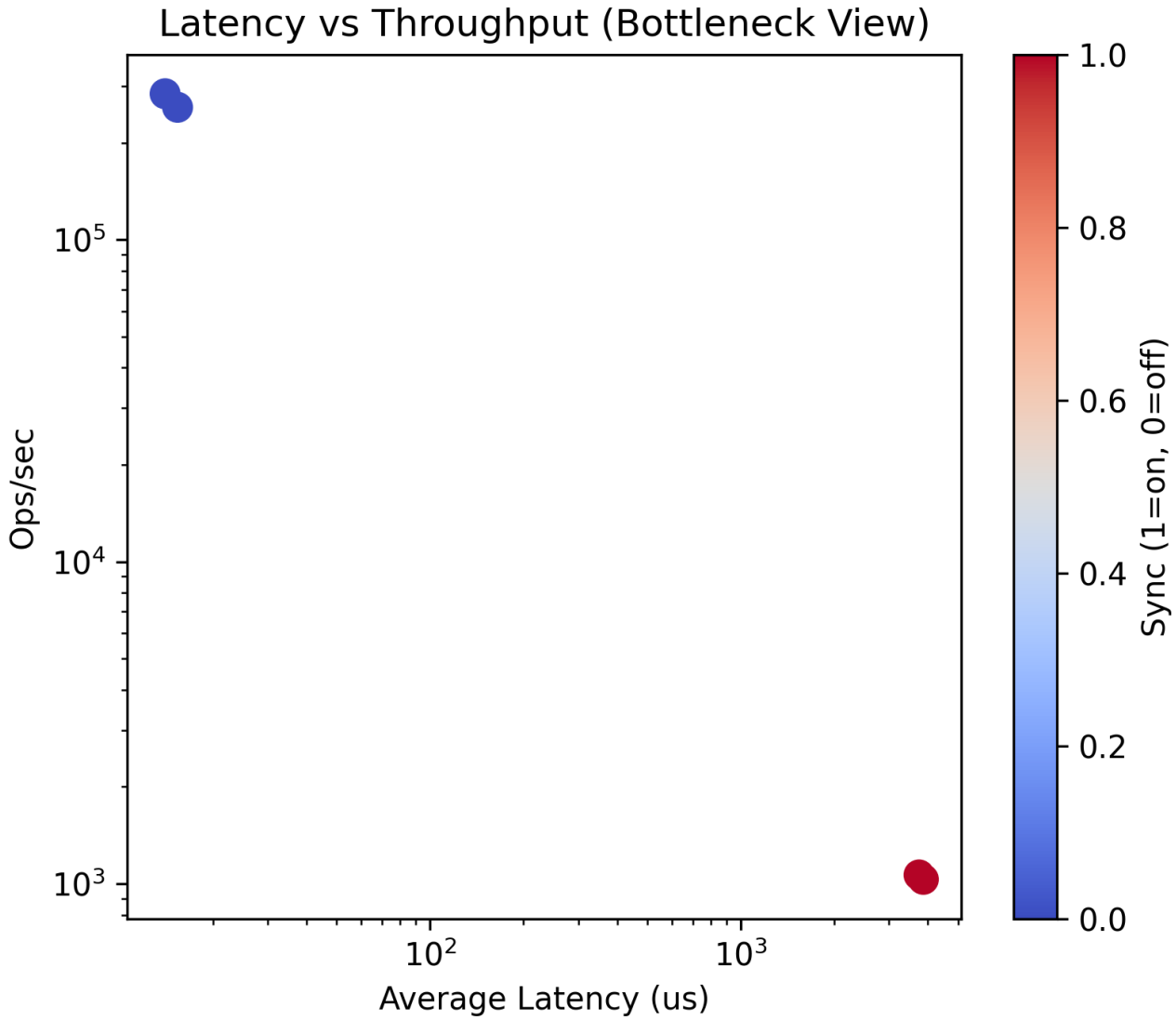



**sync=on에서 throughput이 급락. fsync가 병목으로 지배.**

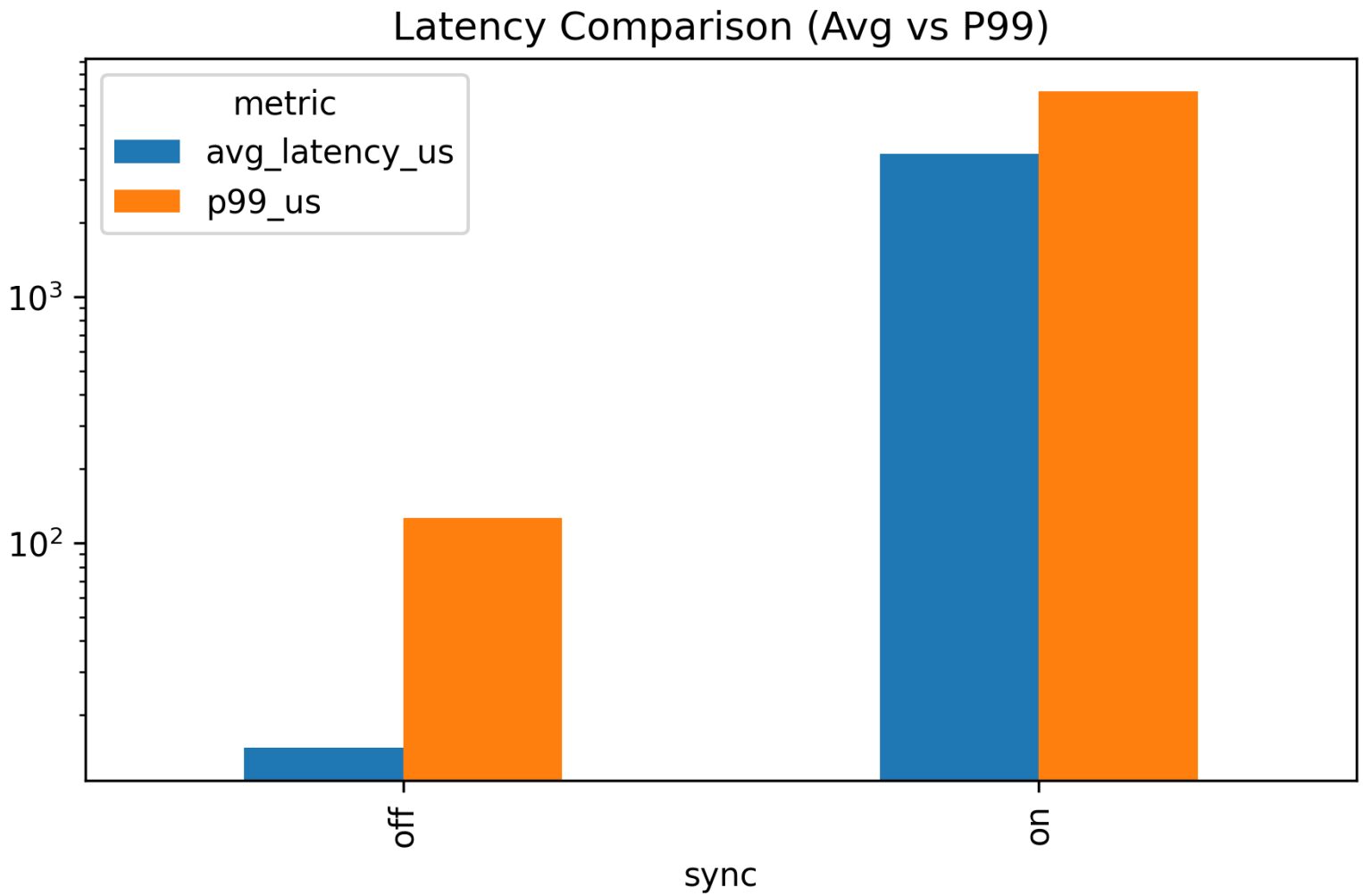
# 결과 2 - ops/sec (fillrandom vs overwrite)



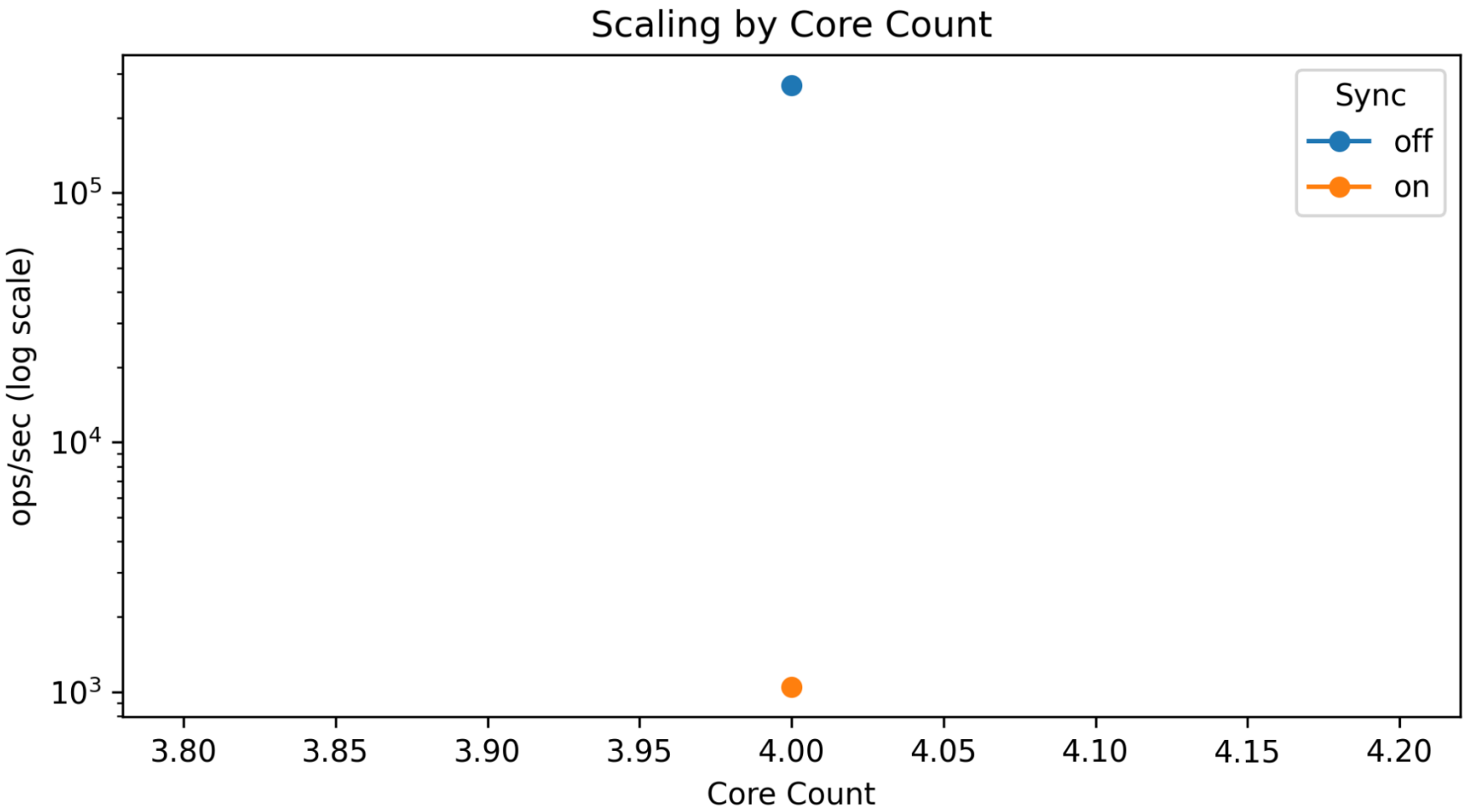
# 결과 3 – Latency vs Throughput



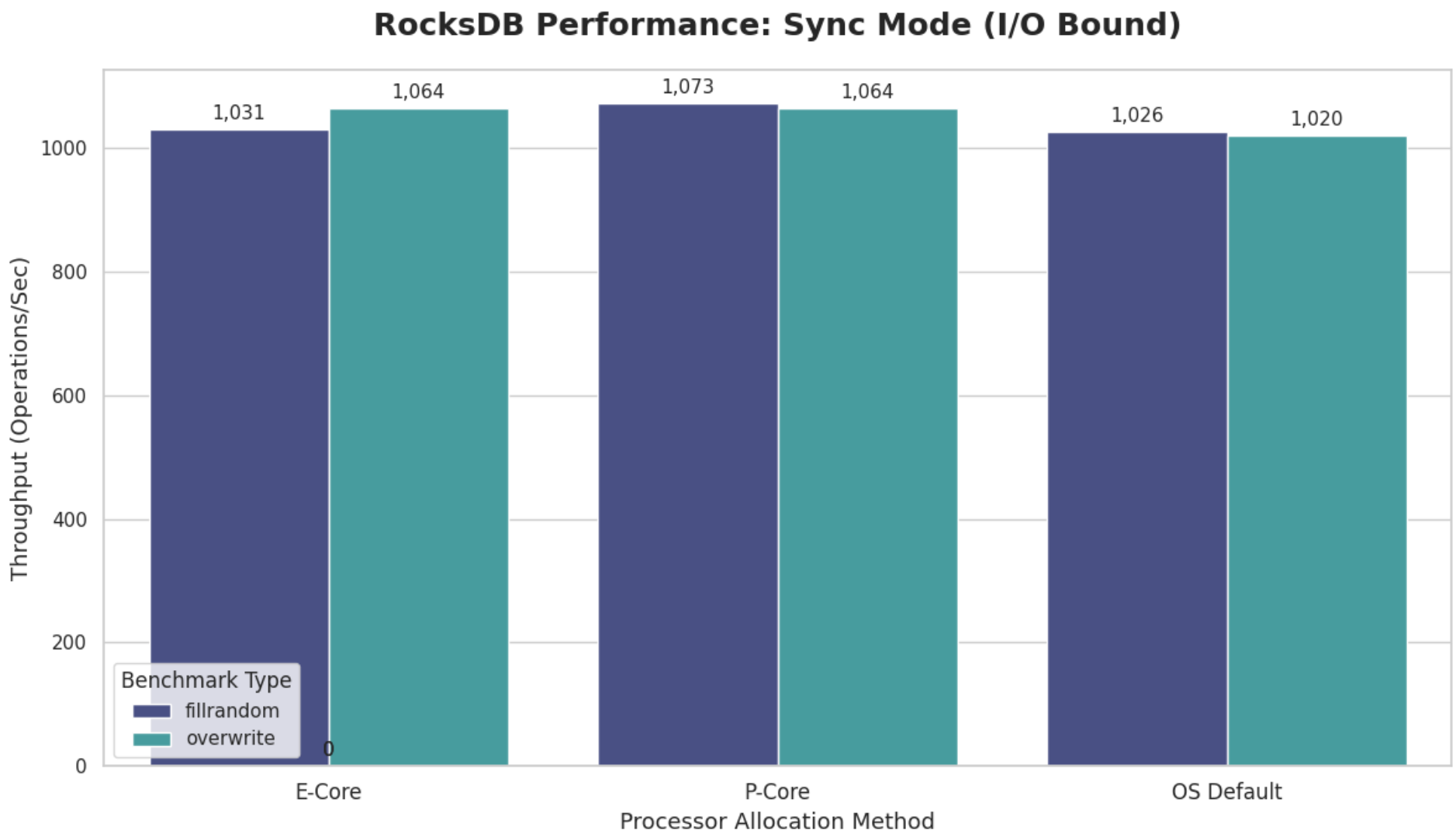
# 결과 4 – Latency Comparison



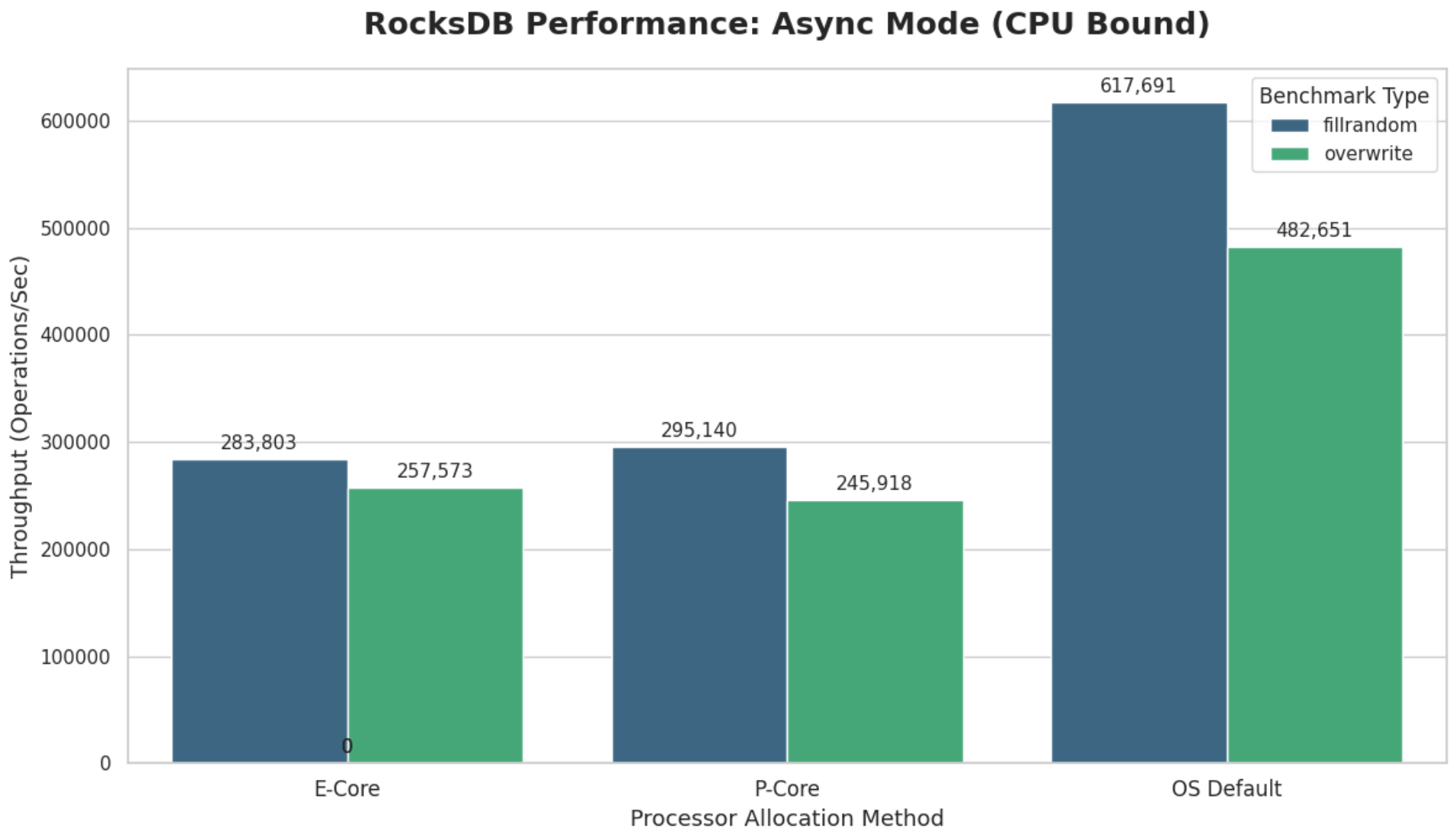
# 결과 5 – Wal fsync 비활성/활성 처리량 비교



# 결과 6 – 동기 처리 방식에서 프로세서별 성능 비교



# 결과 7 – 비동기 처리 방식에서 프로세서별 성능 비교



# 결론

- sync=true는 fsync 대기로 I/O bound가 되며 throughput/latency 상한이 디스크 지연에 의해 결정
- sync=false는 CPU/스케줄링 영향이 커지고, 이기종 코어 특성(P/E) 차이가 드러나기 쉬움
- 호스트 OS인 윈도우에서 스케줄링을 통해 유휴 자원을 효과적으로 자율적으로 동원함
- 코어의 고정은 통제에는 유리하지만, OS의 동적 분산/유휴 자원 동원을 막아 성능을 제한할 수 있음
- 설정 없음 > P > e 코어 순으로 성능의 차이가 발생함

# Thank you