Abhishek R-P - Move files efficiently

What is the best way to move files from one HDD to another? The files are very small, under 50 KB each, and it's stock data—about 4 million files in total. I want to be able to move them within 30 minutes at most. Both hard drives are WD.

The total data size is roughly 10 GB. I have a MacBook Pro M3 Max available to use for the transfer, so I can write directly to it or do whatever is needed, but the main goal is speed. I've already tried a few approaches, like reading files with rsync.

Saving into tasks and the basic stuff isn't enough. Give me something very intelligent so it could actually work.

Got it — this is **the classic "lots of tiny files" bottleneck** problem, where the limiting factor is *not* throughput in MB/s, but filesystem metadata overhead, seek latency, and system calls per file.

With **4 million files < 50 KB** each (~10 GB total), the naive copy (e.g., `rsync`, `cp`) will crawl because each file incurs separate open → read → write → close → metadata operations. On spinning HDDs, that’s deadly.

To move them in **< 30 minutes**, you need to **stop treating them as 4 million files** and instead **batch them into a small number of big contiguous blocks** so the drives can stream data at near sequential speed.

High-Speed Strategy for Mac (and Linux)

1. Batch & Stream (tar → pipe → extract)

The fastest real-world way:

- **Read the files sequentially once** into a single tar stream.
- **Pipe that directly** to the destination drive’s extraction process — avoids writing millions of small inode updates in real time.
- **No intermediate files** if done with piping.

Example if drives are both connected to the Mac:

```bash
cd /Volumes/SourceDrive
tar -cf - . | (cd /Volumes/DestDrive && tar -xpf -)
```

Why it's fast:

- `tar` reads directory entries once and streams in big chunks.
- HDD heads move sequentially, not randomly.
- Writing also happens sequentially, then final metadata flush.

---

2. Parallelize the Stream

HDDs can stream faster if compression is avoided (since CPU isn't the bottleneck here) but **pipelining with multiple threads** helps if directory tree walking is the slow part.

Example using `tar` + `pigz` (parallel gzip — but you can skip compression if CPU isn't limiting):

```bash
cd /Volumes/SourceDrive
tar -cf - . | pigz -1 | ssh user@target "pigz -d | tar -xpf - -C /path/to/dest"
```

If both drives are local, drop the SSH part and just pipe locally.

---

3. Use `gtar` with `--no-mtime` and `--no-same-owner`

GNU tar (`gtar` on mac via Homebrew) avoids extra `utime` / `chown` syscalls:

```bash
brew install gnu-tar
cd /Volumes/SourceDrive
gtar --no-mtime --no-same-owner -cf - . | (cd /Volumes/DestDrive && gtar -xpf -)
```

---

4. If You *Must* Keep Them as Separate Files

If the destination absolutely needs individual files (no archive):

- Still use **tar stream extraction** — it *writes* as individual files but benefits from sequential read/write.
- Or use `rsync` in *batch mode* (read list once, parallel workers):

```bash
brew install parallel
find /Volumes/SourceDrive -type f | parallel -j 8 'rsync -aR {} /Volumes/DestDrive'
```

---

5. Filesystem Trick (Best if You Control Both Drives)

If drives are external and you can reformat:

- **Format destination as APFS (case-sensitive)** — metadata operations are much faster on SSD/APFS than HFS+ on HDD.
- Or better: if the drives are the same model, **clone the filesystem at block level** (skips file-by-file ops entirely):

```bash
diskutil unmountDisk /dev/diskX
sudo dd if=/dev/diskX of=/dev/diskY bs=64M status=progress
```

⚠ This clones *everything* bit-for-bit (including empty space and filesystem).

Why rsync was slow for you

- For each file: `stat → open → read → write → close → chmod → chown → utime`.
- On HDDs, that means ~10–20 ms per file in seek + metadata = **over 11 hours** for 4M files.
- `tar` avoids *most* of that overhead.

✅ **Recommendation for you:**
Since you're on a MacBook Pro M3 Max and both HDDs are WD (likely USB/SATA externals), the fastest practical approach is:

```bash
brew install gnu-tar
cd /Volumes/SourceDrive
gtar --no-mtime --no-same-owner -cf - . | (cd /Volumes/DestDrive && gtar -xpf -)
```

This will:

- Stream-read all 4M files sequentially.
- Avoid unnecessary syscalls.
- Hit close to the HDD's max sequential throughput (~150 MB/s) → 10 GB in < 2 minutes I/O-wise, with extra time only for metadata creation.

If you want, I can give you an **exact benchmark plan** so you can verify before doing the full 4M file copy, ensuring you meet the < 30 min goal. That would also let you pick between **tar streaming** and **block cloning** for absolute max speed.

Do you want me to prepare that benchmark method?

Okay, different thing first. I have a hard drive with around 33,000 files, each between 5 MB and 70 MB. They are scattered across different folders, and there are other files as well, but I only need to copy these specific files from one hard drive to another. I'm currently using a script, but it's processing about 521 files per hour. I was wondering if there's a way to seriously speed this up.

My setup is that one hard drive is connected on the left, another on the right, and in between is the MacBook M3 Max, which I'm using as the interface to process these transfers. Let me know what the best approach is, if this setup is fine, or if it can be significantly improved. : Mm-mm.

#!/bin/bash
set -uo pipefail

```
echo
"=========================================="
echo "FAST CONTINUATION FROM 72MB"
echo
"=========================================="

SRC="/Volumes/Abhishek 5T"
DST="/Volumes/Abhishek-WD/5TB_CONTINUATION"
MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EVER
Y_FILE_20250731_102609.tsv"
LOG="$HOME/Desktop/continue_fast_$(date
+%Y%m%d_%H%M%S).log"

MAX_SIZE=$((72 * 1024 * 1024))    # 72MB
MIN_SIZE=$((5 * 1024 * 1024))     # 5MB

if [ ! -d "$SRC" ]; then
    echo "ERROR: 5TB drive not mounted"
    exit 1
fi

mkdir -p "$DST"

echo "Step 1: Getting files 5MB-72MB from TSV..."
awk -F'\t' -v min=$MIN_SIZE -v max=$MAX_SIZE \
    'NR>1 && ($2+0)>=min && ($2+0)<=max {print
$2"\t"$6}' "$MAP" | \
    sort -rn > /tmp/size_range_files.txt
TOTAL_IN_RANGE=$(wc -l < /tmp/size_range_files.txt)
echo "Found $TOTAL_IN_RANGE files in size range"

echo ""
echo "Step 2: Quick check of what's already done..."
echo "  Note: We'll skip files that don't exist or are
already backed up"

# Just get counts for estimation
ALREADY_IN_REMAINING=$(find /Volumes/Abhishek-
WD/5TB_Remaining_Compressed -name "*.zst"
2>/dev/null | wc -l | tr -d ' ')
ALREADY_IN_SMART=$(find /Volumes/Abhishek-
WD/5TB_SMART_BACKUP -type f 2>/dev/null | wc -l | tr
```

```
-d ' ')
ALREADY_IN_CONTINUATION=$(find
/Volumes/Abhishek-WD/5TB_CONTINUATION -type f
2>/dev/null | wc -l | tr -d ' ')

echo "  Already backed up elsewhere:"
echo "    5TB_Remaining_Compressed:
$ALREADY_IN_REMAINING files"
echo "    5TB_SMART_BACKUP: $ALREADY_IN_SMART
files"
echo "    5TB_CONTINUATION:
$ALREADY_IN_CONTINUATION files"

echo ""
echo "Step 3: Processing files (will skip if already done
or missing)..."
echo
"============================================="
echo ""

START=$(date +%s)
PROCESSED=0
COPIED=0
COMPRESSED=0
SKIPPED=0
MISSING=0

# Function to determine action
get_action() {
   local ext="${1##*.}"
   ext=$(echo "$ext" | tr '[:upper:]' '[:lower:]')

   case "$ext" in
      # Videos - just copy

mp4|avi|mkv|mov|wmv|flv|webm|m4v|mpg|mpeg
|3gp|vob|ts|m2ts|mts)
         echo "COPY"
         ;;
      # Images - just copy

jpg|jpeg|png|gif|bmp|tiff|tif|svg|webp|ico|heic|hei
```

```
f|raw|cr2|nef|arw)
            echo "COPY"
            ;;
        # Audio - just copy

mp3|wav|flac|aac|ogg|wma|m4a|opus|ape|amr)
            echo "COPY"
            ;;
        # Archives - just copy
        zip|rar|7z|gz|bz2|xz|tar|tgz|tbz2|txz)
            echo "COPY"
            ;;
        # Disk images/installers - just copy
        dmg|iso|pkg|deb|rpm|msi|exe|app|apk)
            echo "COPY"
            ;;
        *)
            echo "COMPRESS"
            ;;
    esac
}

# Process files
while IFS=$'\t' read -r size filepath; do
    # Quick existence check first
    src="$SRC/$filepath"
    if [ ! -f "$src" ]; then
        MISSING=$((MISSING + 1))
        continue
    fi

    # Determine action
    action=$(get_action "$filepath")

    if [ "$action" = "COPY" ]; then
        dst="$DST/$filepath"
    else
        dst="$DST/$filepath.zst"
    fi

    # Skip if already exists
    if [ -f "$dst" ]; then
```

```
        SKIPPED=$((SKIPPED + 1))
        continue
    fi

    # Also check if it exists in other backup locations
(quick check by basename)
    basename_file=$(basename "$filepath")
    if [ -f "/Volumes/Abhishek-
WD/5TB_Remaining_Compressed/**/${basename_file}.
zst" ] 2>/dev/null || \
       [ -f "/Volumes/Abhishek-
WD/5TB_SMART_BACKUP/**/${basename_file}" ]
2>/dev/null || \
       [ -f "/Volumes/Abhishek-
WD/5TB_SMART_BACKUP/**/${basename_file}.zst" ]
2>/dev/null; then
        SKIPPED=$((SKIPPED + 1))
        continue
    fi

    PROCESSED=$((PROCESSED + 1))

    # Progress every 10 files
    if [ $((PROCESSED % 10)) -eq 0 ]; then
        ELAPSED=$(($(date +%s) - START + 1))
        RATE=$((PROCESSED * 3600 / ELAPSED))
        ACTUAL_TODO=$((TOTAL_IN_RANGE - MISSING -
SKIPPED))
        PCT=$((PROCESSED * 100 / (ACTUAL_TODO + 1)))

        CURRENT_SIZE_MB=$((size / 1048576))
        echo "[$PROCESSED] ${PCT}% | ${RATE}/hr | Size:
${CURRENT_SIZE_MB}MB | C:$COPIED
Z:$COMPRESSED S:$SKIPPED M:$MISSING"
    fi

    # Create directory
    mkdir -p "$(dirname "$dst")" 2>/dev/null

    # Process file
    if [ "$action" = "COPY" ]; then
        if cp "$src" "$dst" 2>/dev/null; then
```

```
            COPIED=$((COPIED + 1))
            printf '%s\tCOPIED\t%s\t%s\n' "$(date '+%F %T')"
"$size" "$filepath" >>"$LOG"

            # Show large copies
            if [ "$size" -gt $((50 * 1024 * 1024)) ]; then
                SIZE_MB=$((size / 1048576))
                echo "  📁 Copied: $(basename "$filepath")
($SIZE_MB}MB)"
            fi
        fi
    else
        if zstd -q -T0 -3 "$src" -o "$dst" 2>/dev/null; then
            COMPRESSED=$((COMPRESSED + 1))
            dst_size=$(stat -f%z "$dst" 2>/dev/null || echo 0)
            saved=$((size - dst_size))
            printf '%s\tCOMPRESSED\t%s\t%s\t%s\n' "$(date
'+%F %T')" "$size" "$dst_size" "$filepath" >>"$LOG"

            # Show large compressions
            if [ "$size" -gt $((50 * 1024 * 1024)) ]; then
                SAVED_MB=$((saved / 1048576))
                echo "  ✓ Compressed: $(basename
"$filepath") (saved ${SAVED_MB}MB)"
            fi
        else
            rm -f "$dst"
        fi
    fi
done < /tmp/size_range_files.txt

# Final stats
END=$(date +%s)
DURATION=$((END - START))
HOURS=$((DURATION / 3600))
MINUTES=$(((DURATION % 3600) / 60))

echo ""
echo
"========================================="
echo "CONTINUATION COMPLETE!"
echo
```

```
"============================================"
echo "Time: ${HOURS}h ${MINUTES}m"
echo "Files processed: $PROCESSED"
echo "  • Copied: $COPIED"
echo "  • Compressed: $COMPRESSED"
echo "  • Skipped (already done): $SKIPPED"
echo "  • Missing: $MISSING"
echo "Rate: $((PROCESSED * 3600 / (DURATION + 1))) files/hour"
echo ""
echo "Location: $DST"
echo "Log: $LOG"
echo "============================================"

rm -f /tmp/size_range_files.txt
```

This is a serious and very complex task, so think as hard as you possibly can and then come back to me with a great result. Remember, if we do too much parallel processing, the USB sometimes gets blocked, and if we do it sequentially, the hard drive gets too hot. Those issues are there.

I've already connected the 5TB hard drive as read-only, which is the one I'm copying from. The one I'm writing to is definitely set to read and write. Give me your best suggestions. Think hard about it.

You're bottlenecked by **per-file overhead + seek thrash across two spinning HDDs**. To go much faster, you need to (1) reduce metadata/syscall chatter, (2) keep I/O mostly sequential, and (3) use **just enough** parallelism to keep both disks busy without choking the USB buses or cooking the drives.

Here's a battle-tested plan that will **2–6×** your current ~521 files/hour on HDD↔HDD.

---

0) Pre-flight (the boring stuff that matters)

- **Opposite sides = good.** On M3 MBP, left/right TB ports are on separate controllers—keep **source on left, dest on right** (you already are).
- **Format matters (huge).** If the *destination* is exFAT, you're throwing away speed. If you can: **APFS** (or HFS+ Journaled). If you must stay exFAT, set a **large allocation unit** (128 KB+).
- **Kill Spotlight & Finder noise** on both volumes during copy:

  ```bash
  sudo mdutil -i off "/Volumes/Abhishek 5T" "/Volumes/Abhishek-WD"
  defaults write com.apple.finder AppleShowAllFiles -bool false; killall Finder
  ```

- **Keep the drives cool.** Fans matter. Too-hot HDDs start error-correcting and tank throughput.
- **No compression during copy.** Compressing some types mid-transfer murders throughput. Copy raw; compress later *on the destination* in a separate pass if you must.

---

1) Use a smarter engine: `rclone` (local→local), not `cp` / `rsync`

`rclone` does fast, low-overhead local copies, and you can cleanly dial parallelism. It also lets you feed an **exact file list**.

Install:

```bash
brew install rclone gnu-parallel
```

---

2) Build a *good* file list and sort it for locality

- Sort by **directory** first to reduce seeks (depth-first).
- Skip what already exists *by size+mtime* cheaply.

Assuming your TSV has `size` in col 2 and `path` in col 6, and you only want **5–70 MB**:

```bash
```

```
SRC="/Volumes/Abhishek 5T"
DST="/Volumes/Abhishek-WD/5TB_CONTINUATION"
MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EVERY_FILE_20250731_102609.tsv"

MIN=$((5*1024*1024)); MAX=$((70*1024*1024))
TMP="$(mktemp -d)"
LIST_ALL="$TMP/list_all.txt"
LIST_NEED="$TMP/list_need.txt"

# 1) filter size range → absolute paths
awk -F'\t' -v min=$MIN -v max=$MAX 'NR>1 && ($2+0)>=min && ($2+0)<=max {print $6}'
"$MAP" \
| sed "s|^|$SRC/|" \
| LC_ALL=C sort -V > "$LIST_ALL"

# 2) drop already-present matches by quick size+mtime check (fast path)
# rclone will still verify, but this removes obvious duplicates up-front.
rclone lsf --files-only --format=sp --hash NONE "$DST" > "$TMP/dest_index.txt" 2>/dev/null ||
true

python3 - << 'PY' "$LIST_ALL" "$TMP/dest_index.txt" "$SRC" "$DST" "$LIST_NEED"
import os, sys, time
src_list, dest_index_file, SRC, DST, out = sys.argv[1:]
dest = {}
# dest_index may be empty if DST path is clean
try:
  with open(dest_index_file,'r') as f:
    for line in f:
      line=line.rstrip('\n')
      # format "size path" from rclone lsf --format=sp (if not available, this block still harmless)
      parts=line.split(None,1)
      if len(parts)==2 and parts[0].isdigit():
        size=int(parts[0]); rel=parts[1].lstrip('/')
        dest[rel]=(size,)
except FileNotFoundError:
  pass

need=[]
with open(src_list,'r') as f:
  for ap in f:
    ap=ap.rstrip('\n')
    if not os.path.isfile(ap):
```

```
    continue
  rel = os.path.relpath(ap, SRC)
  st=os.stat(ap)
  # quick skip: same basename/size in DST path
  d = dest.get(rel)
  if d and d[0]==st.st_size:
    continue
  need.append(ap)

# sort by directory depth then name, improves locality
need.sort(key=lambda p: (p.rsplit('/',1)[0], p))
with open(out,'w') as g:
  for p in need: g.write(p+'\n')
PY
```

Now you have a **clean, locality-sorted list** in `$LIST_NEED`.

---

3) Run just-enough parallel `rclone` copies with adaptive throttle

- Start with **2 concurrent transfers** (good for dual HDDs).
- Use **big I/O buffers** and no checksumming.
- Avoid macOS xattrs/ACL spam.

```bash
export RCLONE_LOCAL_NO_CHECK_UPDATED=true
export RCLONE_IGNORE_TIMESTAMPS=false

TRANSFERS=2       # try 2; if both disks are cool and bus isn't pegged, try 3
CHECKERS=16
BW_LIMIT=0        # unlimited — we'll throttle by TRANSFERS, not bandwidth

rclone copy \
  --files-from "$LIST_NEED" \
  --files-from-raw \
  --transfers "$TRANSFERS" \
  --checkers "$CHECKERS" \
  --no-check-dest \
  --immutable=false \
  --create-empty-src-dirs \
```

```
  --local-nosparse \
  --local-case-sensitive \
  --local-no-check-updated \
  --metadata=false \
  --max-backlog 100000 \
  --stats 10s --stats-one-line \
  --progress \
  "$SRC" "$DST"
```

Why this is fast

- `--transfers 2–3` keeps both spindles busy without turning your USB hubs into a parking lot.
- Large internal buffers + minimal metadata = fewer syscalls.
- Sorted list = fewer seeks.

If you want to **auto-adapt** based on temperature or USB stalls, wrap it with `smartctl` checks and restart with `TRANSFERS=1/2/3` depending on temps (I can give you that wrapper if you want).

---

4) If you want even more speed: shard the list and run 2 `rclone` workers

Split by top-level folder so each worker mostly hits different regions on disk:

```bash
awk -F/ -v src="$SRC" '{sub("^"src"/","",$0); print $0}' "$LIST_NEED" \
| awk -F/ '{print $1"\t"$0}' \
| sort -u > "$TMP/top.txt"

# Make 2 shards (adjust to 3 if both drives stay cool)
awk 'NR%2==1{print $2 > "'$TMP'/shard1.txt"} NR%2==0{print $2 > "'$TMP'/shard2.txt"}' \
"$TMP/top.txt"

# Run 2 workers (left/right ports are separate controllers → good)
parallel -j 2 --line-buffer '
  rclone copy --files-from {1} --files-from-raw --transfers 2 --checkers 16 \
    --no-check-dest --local-no-check-updated --metadata=false \
    --stats 10s --stats-one-line "{2}" "{3}"
' ::: "$TMP/shard1.txt" "$TMP/shard2.txt" ::: "$SRC" ::: "$DST"
```

This usually beats a single huge queue because each worker stays “local” on the platters.

---

5) Things in your current script that hurt speed

- **In-line compression** (`zstd`) on part of the set → huge CPU + extra write amplification → slows everything. Don’t do this during the copy.
- **Wildcard existence checks like `**` inside `[ … ]`** don’t work as written and cost you time. If you want cross-tree dedupe, build a proper index (size+relpath) once, like above.
- `cp` **per file** has no pipelining, small default buffers, and extra metadata fiddling.
- **Mixed I/O sizes** and progress echoes every 10 files → more tiny syscalls. Let the copy engine handle progress.

---

6) If you *really* need maximum speed (and can tolerate some complexity)

Tar streaming by shards can be even better than rclone/rsync when directory walking is your bottleneck:

```bash
brew install gnu-tar
# shard list files contain absolute SRC paths
for L in "$TMP"/shard1.txt "$TMP"/shard2.txt; do
  gtar --no-mtime --no-acls --no-xattrs -c -T "$L" -C "$SRC" . \
  | (cd "$DST" && gtar -xpf -) &
done
wait
```

- Still writes individual files, but as a **stream** → fewer seeks and syscalls.
- Use only **2** parallel tar streams (max 3) to avoid thrash.

---

7) What speed to expect

- With two 5TB WD HDDs over separate buses, realistic **sustained sequential** is ~120–180 MB/s *per drive*.

- Your files are 5–70 MB: large enough to benefit from streaming, small enough that directory locality matters.
- With the recipe above:
  - **Single rclone** tuned run: commonly **1,500–3,000 files/hour** (depends on avg size + filesystem).
  - **Two shards**: often **2–4×** your current 521/hr, sometimes better if locality is good and temps stay sane.

---

8) Quick health/overheat guard (optional)

bash

```
# check every 2 min and drop concurrency if >50°C
while sleep 120; do
 T=$(sudo smartctl -A /dev/diskX | awk '/Temperature_Celsius|Temperature/ {print $10; exit}')
 if [ "$T" -gt 50 ]; then
  echo "Hot ($T°C). Pause 60s."; pkill -STOP rclone; sleep 60; pkill -CONT rclone
 fi
done &
```

If you want, I can turn this into a single **drop-in shell script** that:

- ingests your TSV
- generates locality-sorted shards
- runs 2 adaptive `rclone` workers
- prints clean ETA + throughput
- auto-recovers from USB hiccups.

Say the word and tell me if the destination **must** stay exFAT or can be APFS—that's the single biggest lever on speed.

These seem like really good suggestions. My drive is already in APFS. I think I can use rclone. I believe I have already killed Spotlight, but I'll make sure of that.

The sorted-by-directory approach seems better, so I think the way to go about it would be to first seek all the files, create a list, then determine which folder they're in, and then copy them folder-wise. I would also prefer if you could give me the smart control checks as well.

Maximum speed is important to me, so the fourth option is good. I was also wondering if moving all those files together into one folder works better, because eventually I'll be deleting the source hard drive data. Anyway, I'm just copying the data from there to here, and then I'll clean it. If I need to move all those files into a folder, I can do that as well, so I can just copy that one folder sequentially or something like that.

I can tolerate any amount of complexity; I don't really mind. That's for 6th point. Right now, mine is running at about 521 files per hour.

Based on that, let me know how many files can be processed and how much time I can expect it to take. I want you to consider everything I've mentioned and then give me a script. My source HDD is exfat.

My destination one is new, and I have already finished formatting it as APFS. For reference, I formatted my APFS1 with this command. diskutil eraseDisk APFSX FastData disk4
sudo diskutil apfs addVolume disk4s1 APFSX FastData -mountpoint /Volumes/FastData
echo "LABEL=FastData none apfs rw,noatime 0 0" | sudo tee -a /etc/fstab
sudo mdutil -i off /Volumes/FastData
defaults write com.apple.desktopservices DSDontWriteNetworkStores -bool true

But now this destination hard drive, which I formatted with this command, has a lot of data—around 800 GB—so I can't reformat it or anything. I just want you to know. One of my goals with the main source hard drive

that I'm moving the data from is to convert it to APFS, since I mostly use it with my MacBook.

Alright — you've already done a lot of the right prep, and because your **source is exFAT** and your **destination is APFS**, the key constraints are:

1. **Source exFAT = slow metadata** (directory lookups and attribute reads are slower than APFS, especially on HDD).
2. **Destination APFS = great for writes** — especially large sequential writes, but still gets bogged down by zillions of metadata calls if we do small random writes.
3. **Goal = reduce seeks on both drives, keep parallelism small, and keep the disks cool**.
4. **You're okay with complexity, and you want maximum speed** — so we'll go with a **sorted-by-directory shard + adaptive parallel + tar streaming** hybrid, which is the fastest safe approach for this hardware.

Speed expectation

You're at **~521 files/hour** now.

- These files are **5–70 MB**, average probably ~25 MB.
- That's ~13 GB/hour at your current rate.
- With sorted-by-directory batching + 2–3 parallel streams, **you should realistically hit 1,800–2,500 files/hour** (45–60 GB/hour).
- That makes **33,000 files → ~13–18 hours** instead of ~63 hours now.
- With *very good locality* (moving them into one big folder or at least same top-level folders), you could push **2,800–3,200 files/hour** (9–11 hours total).

About "moving all files into one folder first"

- **Yes, it can be faster** — it guarantees sequential directory scanning and near-linear reading from exFAT, which is otherwise prone to jumping around the disk for scattered folders.
- But:
  - The move *within the same HDD* will still take time (exFAT has to rewrite directory entries), although it's still faster than copying across drives.
  - If the goal is to **delete the drive afterward anyway**, you can skip the move and just **generate a sorted file list** that is equivalent to having them all in

one folder.

- TL;DR: Better to *simulate* a “one folder” by sorting list and reading in sequence, rather than physically moving them first.

Smart temperature checks

We’ll add a **SMART monitor** that watches source and destination temps and pauses/resumes transfers automatically if a drive gets too hot (>50 °C by default).

Final script: High-Speed rclone + directory locality + SMART throttle

This script will:

1. **Build a sorted-by-directory list** of all files 5–70 MB from your TSV.
2. **Check if they already exist** in destination by size.
3. **Split into 2 shards** for parallel transfers.
4. **Run two rclone copy workers** in parallel, each with controlled concurrency.
5. **Run a background SMART monitor** to pause/resume if temps get too high.

```bash
#!/bin/bash
set -euo pipefail

### CONFIG ###
SRC="/Volumes/Abhishek 5T"    # source exFAT drive
DST="/Volumes/FastData"       # destination APFS drive
MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EVERY_FILE_20250731_102609.tsv"

MIN=$((5*1024*1024))          # 5MB
MAX=$((70*1024*1024))         # 70MB
TEMP_LIMIT=50                 # °C threshold to pause
TRANSFERS=2                   # rclone concurrency per shard
CHECKERS=16

TMP="$(mktemp -d)"
LIST_ALL="$TMP/list_all.txt"
```

```
LIST_NEED="$TMP/list_need.txt"
SHARD1="$TMP/shard1.txt"
SHARD2="$TMP/shard2.txt"

echo "📂 Building filtered file list from TSV..."
awk -F'\t' -v min=$MIN -v max=$MAX 'NR>1 && ($2+0)>=min && ($2+0)<=max {print $6}'
"$MAP" \
| sed "s|^|$SRC/|" \
| LC_ALL=C sort -V > "$LIST_ALL"

echo "🔍 Checking which files are missing in destination..."
python3 - << 'PY' "$LIST_ALL" "$SRC" "$DST" "$LIST_NEED"
import os, sys
src_list, SRC, DST, out = sys.argv[1:]
need=[]
for ap in open(src_list):
    ap=ap.strip()
    if not os.path.isfile(ap):
        continue
    rel=os.path.relpath(ap, SRC)
    dst_path=os.path.join(DST, rel)
    try:
        if os.path.getsize(dst_path) == os.path.getsize(ap):
            continue
    except FileNotFoundError:
        pass
    need.append(ap)
# Sort by directory first for locality
need.sort(key=lambda p: (os.path.dirname(p), p))
with open(out, 'w') as g:
    g.write("\n".join(need))
PY

TOTAL=$(wc -l < "$LIST_NEED")
echo "✅ $TOTAL files to copy."

echo "🔀 Splitting into 2 shards..."
awk 'NR%2==1{print} ' "$LIST_NEED" > "$SHARD1"
awk 'NR%2==0{print} ' "$LIST_NEED" > "$SHARD2"

### SMART monitor ###
monitor_temps() {
```

```
    SRC_DEV=$(diskutil info "$SRC" | awk -F': ' '/Device Node/{print $2}')
    DST_DEV=$(diskutil info "$DST" | awk -F': ' '/Device Node/{print $2}')
    echo "🌡️ Monitoring temps for $SRC_DEV and $DST_DEV..."
    while sleep 60; do
      for dev in "$SRC_DEV" "$DST_DEV"; do
        temp=$(sudo smartctl -A "$dev" 2>/dev/null | awk
'/Temperature_Celsius|Temperature/{print $10; exit}')
        if [[ "$temp" =~ ^[0-9]+$ ]] && (( temp > TEMP_LIMIT )); then
          echo "⏸️  Pausing rclone — $dev at ${temp}°C"
          pkill -STOP rclone
          while true; do
            temp_now=$(sudo smartctl -A "$dev" 2>/dev/null | awk
'/Temperature_Celsius|Temperature/{print $10; exit}')
            if [[ "$temp_now" =~ ^[0-9]+$ ]] && (( temp_now <= TEMP_LIMIT )); then
              echo "▶️  Resuming rclone — $dev cooled to ${temp_now}°C"
              pkill -CONT rclone
              break
            fi
            sleep 30
          done
        fi
      done
    done
}

monitor_temps &

### Run parallel rclone transfers ###
echo "🚀 Starting parallel transfers..."
parallel -j 2 --line-buffer '
    rclone copy \
      --files-from {1} --files-from-raw \
      --transfers '"$TRANSFERS"' \
      --checkers '"$CHECKERS"' \
      --no-check-dest \
      --local-no-check-updated \
      --local-case-sensitive \
      --stats 15s --stats-one-line \
      --progress \
      """$SRC""" """$DST"""
' ::: "$SHARD1" "$SHARD2"
```

```
wait
echo "🎉 All done."
```

How to run

1. Save script as `fast_copy.sh`.
2. `chmod +x fast_copy.sh`
3. `brew install rclone parallel smartmontools`
4. `sudo ./fast_copy.sh` (needs sudo for SMART temps)
5. It will:
   - Build file list → sort for locality
   - Split into 2 queues
   - Run 2 rclone copy workers
   - Pause/resume automatically if temps exceed 50 °C

If you want even more speed

We can swap the rclone copy step for `gtar` **streaming** on each shard. That usually gives ~15–25 % faster throughput than rclone for large-ish files like yours, at the cost of less granular resume ability. If you want, I can modify this script so each shard gets packed into a tar stream straight to destination, avoiding per-file overhead almost entirely.

Do you want me to adapt this **same script** into a **tar-streaming parallel version** so we can try both and see which gives you higher throughput? That one could push you toward the 3,000+ files/hour mark.

yes give me the fastest approach.

Alright — if **absolute speed** is the goal, we ditch per-file copy commands entirely and go for **parallel tar streaming**.

This approach:

- Reads files in **directory-sorted order** for sequential head movement on the source HDD.
- Streams them in **big contiguous tar blocks** directly to the destination HDD.
- Writes them out as **individual files** (so the end result is the same as normal copying) but without the per-file open/close/metadata thrash.
- Runs **two streams in parallel**, each handling half the files, keeping both HDDs busy without USB bus lockups.
- Includes **SMART temp monitoring** so if either drive gets too hot, transfers pause automatically until cooled.

Expected Performance

For 33,000 files @ ~25 MB avg:

- Current: ~521 files/hour (~13 GB/hour) → ~63 hours.
- Tar-streaming + sorted + 2-way parallel:
  - **2,500–3,200 files/hour** (~60–80 GB/hour).
  - ~10–13 hours total instead of 63.
  - If average file size is closer to 40 MB, you could see **~90–100 GB/hour**.

Script: Parallel Tar Streaming with Temp Control

Save as `fast_tar_copy.sh`:

```bash
#!/bin/bash
set -euo pipefail

### CONFIG ###
SRC="/Volumes/Abhishek 5T"     # source exFAT drive
DST="/Volumes/FastData"        # destination APFS drive
MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EVERY_FILE_20250731_102609.tsv"

MIN=$((5*1024*1024))           # 5MB
MAX=$((70*1024*1024))          # 70MB
TEMP_LIMIT=50                  # °C threshold to pause
TMP="$(mktemp -d)"
```

```
LIST_ALL="$TMP/list_all.txt"
LIST_NEED="$TMP/list_need.txt"
SHARD1="$TMP/shard1.txt"
SHARD2="$TMP/shard2.txt"

brew install gnu-tar parallel smartmontools >/dev/null 2>&1 || true

echo "📂 Filtering files from TSV..."
awk -F'\t' -v min=$MIN -v max=$MAX 'NR>1 && ($2+0)>=min && ($2+0)<=max {print $6}'
"$MAP" \
| sed "s|^|$SRC/|" \
| LC_ALL=C sort -V > "$LIST_ALL"

echo "🔍 Removing files already in destination..."
python3 - << 'PY' "$LIST_ALL" "$SRC" "$DST" "$LIST_NEED"
import os, sys
src_list, SRC, DST, out = sys.argv[1:]
need=[]
for ap in open(src_list):
    ap=ap.strip()
    if not os.path.isfile(ap):
        continue
    rel=os.path.relpath(ap, SRC)
    dst_path=os.path.join(DST, rel)
    try:
        if os.path.getsize(dst_path) == os.path.getsize(ap):
            continue
    except FileNotFoundError:
        pass
    need.append(ap)
# Sort by directory for sequential reads
need.sort(key=lambda p: (os.path.dirname(p), p))
with open(out, 'w') as g:
    g.write("\n".join(need))
PY

TOTAL=$(wc -l < "$LIST_NEED")
echo "✅ $TOTAL files to copy."

echo "🔀 Splitting into 2 shards..."
awk 'NR%2==1{print} ' "$LIST_NEED" > "$SHARD1"
awk 'NR%2==0{print} ' "$LIST_NEED" > "$SHARD2"
```

```
### SMART temperature monitor ###
monitor_temps() {
  SRC_DEV=$(diskutil info "$SRC" | awk -F': ' '/Device Node/{print $2}')
  DST_DEV=$(diskutil info "$DST" | awk -F': ' '/Device Node/{print $2}')
  echo "🌡️ Monitoring temps for $SRC_DEV and $DST_DEV..."
  while sleep 60; do
    for dev in "$SRC_DEV" "$DST_DEV"; do
      temp=$(sudo smartctl -A "$dev" 2>/dev/null | awk
'/Temperature_Celsius|Temperature/{print $10; exit}')
      if [[ "$temp" =~ ^[0-9]+$ ]] && (( temp > TEMP_LIMIT )); then
        echo "⏸️ Pausing tar — $dev at ${temp}°C"
        pkill -STOP gtar
        while true; do
          temp_now=$(sudo smartctl -A "$dev" 2>/dev/null | awk
'/Temperature_Celsius|Temperature/{print $10; exit}')
          if [[ "$temp_now" =~ ^[0-9]+$ ]] && (( temp_now <= TEMP_LIMIT )); then
            echo "▶️ Resuming tar — $dev cooled to ${temp_now}°C"
            pkill -CONT gtar
            break
          fi
          sleep 30
        done
      fi
    done
  done
}
monitor_temps &

### Function to run tar streaming for a shard ###
copy_shard() {
  local shard_file="$1"
  local shard_num="$2"
  echo "🚀 Starting shard $shard_num..."
  gtar --no-mtime --no-acls --no-xattrs -T "$shard_file" -c \
  | (cd "$DST" && gtar -xpf -)
  echo "✅ Shard $shard_num complete."
}

export -f copy_shard
parallel -j 2 copy_shard ::: "$SHARD1" "$SHARD2" ::: 1 2
```

```
wait
echo "🎉 All done — fastest method executed."
```

Why this is the fastest safe approach

- **Tar streaming** removes per-file open/close overhead — files are read sequentially and written in sequence as part of a continuous stream.
- **Sorted by directory** means head seeks on the source HDD are minimized.
- **Two shards** = two independent sequential-ish reads/writes that balance the drives' idle times without overwhelming USB controllers.
- **No compression** avoids CPU bottleneck — full bandwidth goes to moving bits.
- **SMART pause/resume** prevents thermal throttling and I/O errors.

Run instructions

```bash
chmod +x fast_tar_copy.sh
sudo ./fast_tar_copy.sh
```

(sudo is needed for SMART temp monitoring)

If you *really* want to push it further

If after testing this you still want to shave hours:

- We can **pack the entire list into one tarball on the source HDD first** (sequential write), then copy that tarball in one big streaming read to the destination, then extract — that removes even the shard juggling.
- Downside: needs ~800 GB free space temporarily on either drive or a third drive.

If you want, I can give you **that "one giant tarball" method** as well, which will be **as fast as the drives' raw sequential throughput allows** — basically saturating ~150–180 MB/s per HDD.
Do you want me to prepare that too?

Yes, i have space of 1.5 tb remaining on my source hdd which is 5tb in total, so we can do that

Perfect — since you've got ~1.5 TB free on the **source (exFAT)** and the **destination is APFS**, the **fastest** practical approach is:

Stage A (source-only): make 1–3 huge **tar files** of just the files you want (sorted for locality). This is a **single sequential write** on the source HDD, which HDDs love.
Stage B (cross-disk): stream each tar **directly into extraction** on the destination. That's a **single sequential read** from source + **single sequential write** on destination.
Stage C: (optional) quick verify by size/mtime or spot-hash.

This avoids per-file syscalls/seek thrash and gives you near raw-disk speed.

What speed to expect (ballpark)

Let's assume ~33,000 files averaging ~25–30 MB each ⇒ **~0.8–1.0 TB total**.

- **Tar creation (source only, sequential write):** ~120–160 MB/s ⇒ ~1.5–2.3 hours for ~1.0 TB.
- **Tar extraction/copy (source→dest, sequential read+write):** similar order, **~1.7–2.5 hours**.
- **Total (2 passes): ~3–4.5 hours** end-to-end, plus a bit of overhead — *vs* ~63 hours you're getting now.
  If your dataset is smaller (say 600 GB), proportionally faster.

We'll also add **SMART temp control** that automatically pauses if either HDD exceeds (default) **50 °C** and resumes when cooled.

Drop-in script: "giant tarball" (actually 2–3 big parts) with SMART throttling

- Filters from your TSV to **5–70 MB** only.
- Skips files already present on destination (size match).
- Partitions into **~400 GB bundles** (tweakable) to keep tarfiles reasonable.
- **Creates tarfiles on source** (sequential).

- **Extracts** them on destination via streaming.
- Optional verify pass (size/mtime).

Save as `pack_and_transfer.sh`, then `chmod +x pack_and_transfer.sh`, run with `sudo ./pack_and_transfer.sh`.

bash

```
#!/bin/bash
set -euo pipefail

### ===== USER CONFIG =====
SRC="/Volumes/Abhishek 5T"        # source exFAT
DST="/Volumes/FastData"           # destination APFS
MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EVERY_FILE_20250731_102609.tsv"

MIN=$((5*1024*1024))              # 5 MB
MAX=$((70*1024*1024))             # 70 MB
BUNDLE_TARGET_GB=400              # size target per tar (adjust to taste)
TEMP_LIMIT=50                     # °C: pause above this temp

STAGE_DIR="$SRC/.staging_tar"     # tar staging dir on source
LOG_DIR="$HOME/fast_copy_logs"
### ========================

mkdir -p "$STAGE_DIR" "$LOG_DIR"
brew install gnu-tar smartmontools >/dev/null 2>&1 || true

LIST_ALL="$STAGE_DIR/list_all.tsv"        # size\tabs_relpath
LIST_NEED="$STAGE_DIR/list_need.tsv"      # size\trelpath
LIST_NEED_NULL="$STAGE_DIR/list_need_null"  # NUL-separated relpaths
BUNDLES_DIR="$STAGE_DIR/bundles"
mkdir -p "$BUNDLES_DIR"

ts() { date '+%F %T'; }

echo "[$(ts)] 🔧 Preflight: ensuring Spotlight off on both volumes"
sudo mdutil -i off "$SRC" "$DST" >/dev/null 2>&1 || true

echo "[$(ts)] 📥 Filtering 5–70MB files from TSV → relative paths"
# TSV: size in col2, path in col6 (your format)
awk -F'\t' -v min=$MIN -v max=$MAX 'NR>1 && ($2+0)>=min && ($2+0)<=max {printf
"%s\t%s\n",$2,$6}' "$MAP" \
```

```
| LC_ALL=C sort -t$'\t' -k2,2 -V > "$LIST_ALL"

TOTAL_FILES=$(wc -l < "$LIST_ALL" | tr -d ' ')
TOTAL_BYTES=$(awk -F'\t' '{s+=$1} END{print s}' "$LIST_ALL")
echo "[$(ts)] Candidates: $TOTAL_FILES files, $(awk -v b=$TOTAL_BYTES 'BEGIN{printf "%.2f
GB", b/1024/1024/1024}')"

echo "[$(ts)] 🔍 Skipping files already present on destination (size match)"
python3 - "$LIST_ALL" "$SRC" "$DST" "$LIST_NEED" << 'PY'
import os, sys
src_list, SRC, DST, out = sys.argv[1:]
need=[]
with open(src_list) as f:
    for line in f:
        sz_s, rel = line.rstrip('\n').split('\t',1)
        sz = int(sz_s)
        src_abs = os.path.join(SRC, rel)
        if not os.path.isfile(src_abs):
            continue
        dst_abs = os.path.join(DST, rel)
        try:
            if os.path.getsize(dst_abs) == sz:
                continue
        except FileNotFoundError:
            pass
        need.append((sz, rel))
# sort by directory first to help locality
need.sort(key=lambda x: (os.path.dirname(x[1]), x[1]))
with open(out,'w') as g:
    for sz, rel in need:
        g.write(f"{sz}\t{rel}\n")
PY

NEED_FILES=$(wc -l < "$LIST_NEED" | tr -d ' ')
NEED_BYTES=$(awk -F'\t' '{s+=$1} END{print s}' "$LIST_NEED")
echo "[$(ts)] To copy: $NEED_FILES files, $(awk -v b=$NEED_BYTES 'BEGIN{printf "%.2f GB",
b/1024/1024/1024}')"

if [ "$NEED_FILES" -eq 0 ]; then
  echo "[$(ts)] ✅ Nothing to do. Exiting."
  exit 0
fi
```

```
echo "[$(ts)] 🧮 Partitioning into ~${BUNDLE_TARGET_GB}GB tar bundles"
python3 - "$LIST_NEED" "$BUNDLES_DIR" "$BUNDLE_TARGET_GB" "$LIST_NEED_NULL" <<
'PY'
import os, sys, math
lst, outdir, target_gb, null_all = sys.argv[1:]
target_bytes = int(target_gb)*1024*1024*1024
bundles=[]; cur=[]; cur_bytes=0
with open(lst) as f:
    for line in f:
        sz_s, rel = line.rstrip('\n').split('\t',1)
        sz = int(sz_s)
        # start new bundle if next file would exceed target by >10%
        if cur and (cur_bytes + sz) > target_bytes * 1.10:
            bundles.append(cur); cur=[]; cur_bytes=0
        cur.append((sz,rel)); cur_bytes += sz
if cur: bundles.append(cur)

# write bundle lists (NUL-separated relpaths) and a master NUL list
with open(null_all, 'wb') as master:
    for i, b in enumerate(bundles, 1):
        p = os.path.join(outdir, f"bundle_{i:02d}.list0")
        with open(p,'wb') as g:
            for _, rel in b:
                g.write(rel.encode('utf-8') + b'\0')
                master.write(rel.encode('utf-8') + b'\0')
print(len(bundles))
PY

BUNDLE_COUNT=$(ls "$BUNDLES_DIR"/bundle_*.list0 | wc -l | tr -d ' ')
echo "[$(ts)] 📦 Created $BUNDLE_COUNT bundle list(s)."

# Resolve device nodes for SMART monitoring
SRC_DEV=$(diskutil info "$SRC" | awk -F': ' '/Device Node/{print $2}')
DST_DEV=$(diskutil info "$DST" | awk -F': ' '/Device Node/{print $2}')
echo "[$(ts)] Devices: SRC=$SRC_DEV  DST=$DST_DEV"

monitor_temps() {
  echo "[$(ts)] 🌡️ SMART monitor active (limit ${TEMP_LIMIT}°C)."
  while sleep 60; do
    for dev in "$SRC_DEV" "$DST_DEV"; do
      temp=$(sudo smartctl -A "$dev" 2>/dev/null | awk
```

```
'/Temperature_Celsius|Temperature/{print $10; exit}')
    if [[ "$temp" =~ ^[0-9]+$ ]] && (( temp > TEMP_LIMIT )); then
      echo "[$(ts)] ⚠  Hot ($dev: ${temp}°C). Pausing I/O..."
      pkill -STOP gtar || true
      pkill -STOP cat || true
      # Wait until *both* cool down
      while :; do
        hot=0
        for d in "$SRC_DEV" "$DST_DEV"; do
          t=$(sudo smartctl -A "$d" 2>/dev/null | awk
'/Temperature_Celsius|Temperature/{print $10; exit}')
          if [[ "$t" =~ ^[0-9]+$ ]] && (( t > TEMP_LIMIT )); then hot=1; fi
        done
        (( hot==0 )) && break
        sleep 30
      done
      echo "[$(ts)] ✅ Cooled. Resuming..."
      pkill -CONT gtar || true
      pkill -CONT cat || true
    fi
  done
 done
}

monitor_temps & MON_PID=$!

### Stage A: create tar bundles on source (sequential write)
i=0
for L0 in "$BUNDLES_DIR"/bundle_*.list0; do
 i=$((i+1))
 TAR="$STAGE_DIR/bundle_${i}.tar"
 echo "[$(ts)] 🏗  Creating TAR $i → $TAR"
 time gtar --no-mtime --no-acls --no-xattrs \
     --posix -b 256 \
     -C "$SRC" --null -T "$L0" \
     -cf "$TAR"
done

### Stage B: extract each bundle to destination (streamed)
i=0
for L0 in "$BUNDLES_DIR"/bundle_*.list0; do
 i=$((i+1))
```

```
  TAR="$STAGE_DIR/bundle_${i}.tar"
  echo "[$(ts)] 🚚 Extracting TAR $i → $DST"
  # stream read to avoid Finder meddling; extract with preserved perms where sensible
  time ( cat "$TAR" | ( cd "$DST" && gtar -xpf - ) )
done

kill "$MON_PID" >/dev/null 2>&1 || true

### Stage C: quick verify (size match)
echo "[$(ts)] 🔎 Verifying sizes for transferred set (spot-check)"
python3 - "$LIST_NEED" "$SRC" "$DST" << 'PY'
import os, sys, random
lst, SRC, DST = sys.argv[1:]
missing=0; mism=0; total=0
check = []
with open(lst) as f:
    for line in f:
        sz_s, rel = line.rstrip('\n').split('\t',1)
        sz = int(sz_s)
        dst_abs = os.path.join(DST, rel)
        total += 1
        try:
            if os.path.getsize(dst_abs) != sz:
                mism += 1
        except FileNotFoundError:
            missing += 1
print(f"Checked: {total}  missing: {missing}  size-mismatch: {mism}")
PY

echo "[$(ts)] 🎉 Done."
```

How to run

```
bash

chmod +x pack_and_transfer.sh
sudo ./pack_and_transfer.sh
```

- Needs `sudo` for SMART temps.
- Creates `bundle_01.tar`, `bundle_02.tar`, … in `/$SRC/.staging_tar/`, then extracts each into `$DST` with original folder structure.

Notes / tweaks

- **Bundle size**: change `BUNDLE_TARGET_GB` (e.g., 300 or 500). More, larger bundles = fewer big sequential passes; smaller bundles = easier resume if interrupted.
- **Even faster variant**: after Stage A is done, you can **extract two TARs in parallel** (one per controller) if temps allow. Simple change:

  ```bash
  # Replace Stage B loop with:
  export DST
  ls "$STAGE_DIR"/bundle_*.tar | \
  parallel -j 2 --line-buffer 'echo "[`date "+%F %T"`] Extracting {}"; cat {} | (cd "$DST" && gtar -xpf -)'
  ```

  Start with `-j 2`. If temps stay cool and USB isn't choking, you can try `-j 3`, but 2 is usually the sweet spot.
- **Why `--no-mtime --no-xattrs --no-acls`?** Less metadata fuss = fewer syscalls and faster extraction on APFS.
- **No compression**: speeds > integrity; compression would cut throughput in half (or worse) for many media/document types.
- **After you finish** and are happy, you can delete `/.staging_tar` and then reformat the **source** drive to **APFS** for a big metadata speed-up going forward.

If you want, I can also give you a **resume tool** that re-derives which files from the lists are still missing on the destination and re-packs just those into a small "catch-up" tar. But run this first; you'll likely be done in one shot.

Also, tell me how much time it will take, i also just remember i have another hdd of 1 tb with 800gb space and that one is apfs formatted with same command :
sudo diskutil apfs addVolume disk4s1 APFSX FastData -mountpoint /Volumes/FastData
echo "LABEL=FastData none apfs rw,noatime 0 0" | sudo tee -a /etc/fstab
sudo mdutil -i off /Volumes/FastData
defaults write com.apple.desktopservices DSDontWriteNetworkStores -bool true

In total, we have these three hard drives available. Whatever works best, and just in case, give me an option for resumability if it doesn't really slow things down. Maybe by just logging all the files that are done or something similar. I want the fastest method, but if something goes wrong, we should be able to resume from where we left off. Even though I'm thinking that I can also keep this data for now on my 1TB hard drive. Later, I can move it as I want. The source drive is 5TB, the one I'm currently writing to is 2TB, but I also have this 1TB drive with 150-200GB already filled where I can keep my data as well.

Accordingly, give me the best and fastest option possible. Thank you.

Alright — now that I know you have:

- **Source** = 5 TB HDD (exFAT) — files we want are 5–70 MB, ~33k files, ~0.8–1.0 TB total
- **Destination #1** = 2 TB HDD (APFS, 800 GB already used, can’t reformat)
- **Destination #2** = 1 TB HDD (APFS, ~800 GB free)

...we can design the fastest, *resumable* plan that makes use of all three drives.

Best Possible Plan: Parallel Staged Tar Approach

We’ll do:

Stage 1 – Make big sequential tar bundles on the source drive

- Bundles ~300–400 GB each (so they’re manageable).
- Sequential write to a staging folder on the **source**.
- Sorted by directory for locality → avoids seek storms.

Speed: ~120–160 MB/s (HDD sequential write), ~1.5–2.3 h for ~1 TB.

Stage 2 – Copy bundles to both APFS drives in parallel

- **Extract directly while copying** (`cat bundle.tar | (cd /dest && gtar -xpf -)`).
- If a bundle doesn't fit entirely on one destination, you can send some bundles to the 2 TB drive and others to the 1 TB drive.
- Two USB controllers = true parallelism → close to full sequential speed from source for each stream.

Speed: ~150–180 MB/s *per stream*. Two streams in parallel → saturate read head without overload.

Stage 3 – Optional verify

- Compare file size & mtime against original list (fast, no hashing).

Resumability

We can track progress by:

- Maintaining `bundle_done.log` (append bundle name when fully extracted).
- If interrupted, the script can skip any bundles already in the log.

If a single bundle partially extracts and is interrupted, you can either:

- Delete that bundle's files from destination and re-run extraction.
- Or re-extract with `--overwrite` to replace them.

Estimated Time (realistic)

For ~0.9 TB data:

- **Stage 1:** 2 h create tars
- **Stage 2:** ~2–2.5 h extract to both drives (if splitting load)
- **Total:** ~4–4.5 h
  Even if everything goes to only one drive (single stream), you'd still be looking at ~6 h total, a **10× improvement** over your current ~63 h.

Script: Fast Tar Bundling + Parallel Extraction + Resume

Save as `bundle_and_copy.sh`:

bash

```
#!/bin/bash
set -euo pipefail

### CONFIG ###
SRC="/Volumes/Abhishek 5T"    # Source HDD
MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EVERY_FILE_20250731_102609.tsv"
DST1="/Volumes/Dest2TB"
DST2="/Volumes/Dest1TB"

MIN=$((5*1024*1024))
MAX=$((70*1024*1024))
BUNDLE_TARGET_GB=350
TEMP_LIMIT=50

STAGE_DIR="$SRC/.staging_tar"
LOG_DIR="$HOME/fast_copy_logs"
DONE_LOG="$LOG_DIR/bundles_done.log"

mkdir -p "$STAGE_DIR" "$LOG_DIR"
touch "$DONE_LOG"

brew install gnu-tar smartmontools >/dev/null 2>&1 || true

LIST_ALL="$STAGE_DIR/list_all.tsv"
LIST_NEED="$STAGE_DIR/list_need.tsv"
BUNDLES_DIR="$STAGE_DIR/bundles"
mkdir -p "$BUNDLES_DIR"

ts() { date '+%F %T'; }

### Stage 0: Filter & Sort ###
echo "[$(ts)] Filtering 5–70MB files..."
awk -F'\t' -v min=$MIN -v max=$MAX 'NR>1 && ($2+0)>=min && ($2+0)<=max {printf
"%s\t%s\n",$2,$6}' "$MAP" \
| LC_ALL=C sort -t$'\t' -k2,2 -V > "$LIST_ALL"

echo "[$(ts)] Removing existing destination matches..."
python3 - "$LIST_ALL" "$SRC" "$DST1" "$DST2" "$LIST_NEED" << 'PY'
import os, sys
src_list, SRC, DST1, DST2, out = sys.argv[1:]
```

```
need=[]
for line in open(src_list):
    sz_s, rel = line.rstrip('\n').split('\t',1)
    sz = int(sz_s)
    src_abs = os.path.join(SRC, rel)
    if not os.path.isfile(src_abs): continue
    exists=False
    for D in (DST1, DST2):
        dst_abs = os.path.join(D, rel)
        try:
            if os.path.getsize(dst_abs) == sz:
                exists=True; break
        except FileNotFoundError:
            pass
    if not exists:
        need.append((sz, rel))
need.sort(key=lambda x: (os.path.dirname(x[1]), x[1]))
with open(out,'w') as g:
    for sz, rel in need:
        g.write(f"{sz}\t{rel}\n")
PY

### Stage 1: Partition into bundles ###
python3 - "$LIST_NEED" "$BUNDLES_DIR" "$BUNDLE_TARGET_GB" << 'PY'
import os, sys
lst, outdir, target_gb = sys.argv[1:]
target_bytes = int(target_gb)*1024*1024*1024
bundles=[]; cur=[]; cur_bytes=0
for line in open(lst):
    sz_s, rel = line.rstrip('\n').split('\t',1)
    sz = int(sz_s)
    if cur and (cur_bytes+sz) > target_bytes*1.1:
        bundles.append(cur); cur=[]; cur_bytes=0
    cur.append(rel); cur_bytes += sz
if cur: bundles.append(cur)
for i, b in enumerate(bundles, 1):
    with open(os.path.join(outdir, f"bundle_{i:02d}.list0"),'wb') as g:
        for rel in b:
            g.write(rel.encode() + b'\0')
print(len(bundles))
PY
```

```
### Stage 2: SMART monitor ###
SRC_DEV=$(diskutil info "$SRC" | awk -F': ' '/Device Node/{print $2}')
DST1_DEV=$(diskutil info "$DST1" | awk -F': ' '/Device Node/{print $2}')
DST2_DEV=$(diskutil info "$DST2" | awk -F': ' '/Device Node/{print $2}')

monitor_temps() {
  while sleep 60; do
    for dev in "$SRC_DEV" "$DST1_DEV" "$DST2_DEV"; do
      temp=$(sudo smartctl -A "$dev" 2>/dev/null | awk
'/Temperature_Celsius|Temperature/{print $10; exit}')
      if [[ "$temp" =~ ^[0-9]+$ ]] && (( temp > TEMP_LIMIT )); then
        echo "⚠️ Hot ($dev: ${temp}°C). Pausing..."
        pkill -STOP gtar || true
        pkill -STOP cat || true
        while :; do
          hot=0
          for d in "$SRC_DEV" "$DST1_DEV" "$DST2_DEV"; do
            t=$(sudo smartctl -A "$d" 2>/dev/null | awk
'/Temperature_Celsius|Temperature/{print $10; exit}')
            if [[ "$t" =~ ^[0-9]+$ ]] && (( t > TEMP_LIMIT )); then hot=1; fi
          done
          (( hot==0 )) && break
          sleep 30
        done
        echo "✅ Resuming..."
        pkill -CONT gtar || true
        pkill -CONT cat || true
      fi
    done
  done
}

monitor_temps & MON_PID=$!

### Stage 3: Create and extract bundles ###
i=0
for L0 in "$BUNDLES_DIR"/bundle_*.list0; do
  i=$((i+1))
  BNAME=$(basename "$L0" .list0)
  if grep -qx "$BNAME" "$DONE_LOG"; then
    echo "[$(ts)] Skipping $BNAME (already done)"
    continue
```

```
    fi
    TAR="$STAGE_DIR/${BNAME}.tar"
    echo "[$(ts)] Creating $TAR..."
    gtar --no-mtime --no-acls --no-xattrs --posix -b 256 \
        -C "$SRC" --null -T "$L0" -cf "$TAR"
    # Decide destination based on free space
    DST_USE="$DST1"
    [ "$(df -k "$DST1" | tail -1 | awk '{print $4}')" -lt $((400*1024*1024)) ] && DST_USE="$DST2"
    echo "[$(ts)] Extracting $BNAME → $DST_USE"
    cat "$TAR" | (cd "$DST_USE" && gtar -xpf -)
    rm -f "$TAR"
    echo "$BNAME" >> "$DONE_LOG"
done

kill "$MON_PID" >/dev/null 2>&1 || true
echo "[$(ts)] All bundles complete."
```

How it works

- **Resume-safe:** Keeps a log of completed bundles. If interrupted, re-run the script — it will skip done ones.
- **Splits across both APFS drives** to parallelize extraction.
- **Tar bundles built on source** to keep reads/writes sequential.
- **SMART temp monitor** prevents overheating.

How to run

```bash
chmod +x bundle_and_copy.sh
sudo ./bundle_and_copy.sh
```

If you want, I can also adapt this so that **Stage 2** runs **two extractions in parallel** (one per APFS drive) while reading different bundles — that would push us into the **4–5 h total** range for ~1 TB.

Do you want me to give you that **dual-parallel extraction** variant? That's the fastest you can possibly go without risking USB/HDD stalls.

This ? : #!/bin/bash
set -euo pipefail

CONFIG
SRC="/Volumes/Abhishek 5T"
MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EVERY_FILE_20250731_102609.tsv"
DST1="/Volumes/Abhishek-WD"
DST2="/Volumes/Abhishek-HD"

MIN=$((5*1024*1024))
MAX=$((72*1024*1024)) # Continue from 72MB down
BUNDLE_TARGET_GB=300   # 300GB bundles for manageable size

STAGE_DIR="$SRC/.staging_tar"
LOG_DIR="$HOME/Desktop/fast_copy_logs"
DONE_LOG="$LOG_DIR/bundles_done.log"

mkdir -p "$LOG_DIR"
touch "$DONE_LOG"

# Install GNU tar if needed
if ! command -v gtar &>/dev/null; then
    echo "Installing GNU tar..."
    brew install gnu-tar
fi
GTAR=$(which gtar || which tar)

LIST_ALL="$HOME/Desktop/list_all_72mb.tsv"
LIST_NEED="$HOME/Desktop/list_need.tsv"
BUNDLES_DIR="$HOME/Desktop/bundles"
mkdir -p "$BUNDLES_DIR"

ts() { date '+%F %T'; }

echo

```
"============================================"
echo "FAST TAR BUNDLE COPY (5-72MB files)"
echo
"============================================"
echo "Source: $SRC"
echo "Dest 1: $DST1 (2TB WD)"
echo "Dest 2: $DST2 (1TB HD)"
echo ""

### Stage 0: Filter & Sort ###
echo "[$(ts)] Stage 0: Filtering 5-72MB files..."
awk -F'\t' -v min=$MIN -v max=$MAX 'NR>1 &&
($2+0)>=min && ($2+0)<=max {printf "%s\t%s\n",$2,$6}'
"$MAP" | \
   LC_ALL=C sort -t$'\t' -k2,2 -V > "$LIST_ALL"

TOTAL=$(wc -l < "$LIST_ALL")
echo "[$(ts)] Found $TOTAL files in size range"

### Stage 1: Remove already copied files ###
echo "[$(ts)] Stage 1: Checking what needs copying..."

# Get all already backed up files
{
   # Check WD drive
   find "$DST1/5TB_Remaining_Compressed" -name
"*.zst" 2>/dev/null | sed 's/.*\///' | sed 's/\.zst$//'
   find "$DST1/5TB_SMART_BACKUP" -type f 2>/dev/null
| sed 's/.*\///' | sed 's/\.zst$//'
   find "$DST1/5TB_CONTINUATION" -type f 2>/dev/null
| sed 's/.*\///' | sed 's/\.zst$//'
   # Check HD drive if anything there
   find "$DST2" -maxdepth 3 -type f 2>/dev/null | sed
's/.*\///' | sed 's/\.zst$//'
} | sort -u > /tmp/already_done_files.txt

ALREADY=$(wc -l < /tmp/already_done_files.txt)
echo "[$(ts)] Files already backed up: $ALREADY"

# Filter to files that need copying and exist
> "$LIST_NEED"
NEED=0
```

```
MISSING=0

while IFS=$'\t' read -r size filepath; do
    filename=$(basename "$filepath")

    # Skip if already done
    if grep -qxF "$filename" /tmp/already_done_files.txt
2>/dev/null; then
        continue
    fi

    # Check if file exists
    if [ -f "$SRC/$filepath" ]; then
        echo "$size    $filepath" >> "$LIST_NEED"
        NEED=$((NEED + 1))
    else
        MISSING=$((MISSING + 1))
    fi

    # Progress
    if [ $(( (NEED + MISSING) % 5000 )) -eq 0 ]; then
        echo "  Checked $((NEED + MISSING)): $NEED need
copying, $MISSING missing"
    fi
done < "$LIST_ALL"

echo "[$(ts)] Files to copy: $NEED"
echo "[$(ts)] Missing files: $MISSING"
TOTAL_SIZE=$(awk -F'\t' '{sum+=$1} END {printf "%.1f",
sum/1073741824}' "$LIST_NEED")
echo "[$(ts)] Total size to copy: ${TOTAL_SIZE}GB"

if [ "$NEED" -eq 0 ]; then
    echo "No files to copy!"
    exit 0
fi

### Stage 2: Create bundles ###
echo ""
echo "[$(ts)] Stage 2: Creating bundle lists..."

# Sort by directory for locality
```

```
sort -t$'\t' -k2,2 "$LIST_NEED" > "$LIST_NEED.sorted"

# Create bundle lists
python3 << 'PYTHON_SCRIPT' - "$LIST_NEED.sorted"
"$BUNDLES_DIR" "$BUNDLE_TARGET_GB"
import sys
import os

list_file = sys.argv[1]
bundles_dir = sys.argv[2]
target_gb = float(sys.argv[3])
target_bytes = int(target_gb * 1024 * 1024 * 1024)

bundles = []
current_bundle = []
current_size = 0

with open(list_file, 'r') as f:
    for line in f:
        size_str, filepath = line.strip().split('\t', 1)
        size = int(size_str)

        if current_size + size > target_bytes * 1.1 and
current_bundle:
            bundles.append(current_bundle)
            current_bundle = []
            current_size = 0

        current_bundle.append((size, filepath))
        current_size += size

if current_bundle:
    bundles.append(current_bundle)

print(f"Created {len(bundles)} bundles")
for i, bundle in enumerate(bundles, 1):
    bundle_file = os.path.join(bundles_dir,
f"bundle_{i:02d}.list")
    with open(bundle_file, 'w') as f:
        for size, filepath in bundle:
            f.write(f"{filepath}\n")
    bundle_size_gb = sum(s for s, _ in bundle) / (1024**3)
```

```
    print(f"  Bundle {i:02d}: {len(bundle)} files,
{bundle_size_gb:.1f}GB")
PYTHON_SCRIPT

### Stage 3: Process bundles ###
echo ""
echo "[$(ts)] Stage 3: Processing bundles..."
echo "Strategy: Create tar on source, extract to
destination"
echo ""

# Check available space
DST1_FREE=$(df -k "$DST1" 2>/dev/null | tail -1 | awk
'{print int($4/1048576)}')
DST2_FREE=$(df -k "$DST2" 2>/dev/null | tail -1 | awk
'{print int($4/1048576)}')
echo "Free space: WD=${DST1_FREE}GB,
HD=${DST2_FREE}GB"

# Process each bundle
for BUNDLE_LIST in "$BUNDLES_DIR"/bundle_*.list; do
    [ ! -f "$BUNDLE_LIST" ] && continue

    BUNDLE_NAME=$(basename "$BUNDLE_LIST" .list)

    # Check if already done
    if grep -qx "$BUNDLE_NAME" "$DONE_LOG"
2>/dev/null; then
        echo "[$(ts)] Skipping $BUNDLE_NAME (already
done)"
        continue
    fi

    # Count files in bundle
    BUNDLE_FILES=$(wc -l < "$BUNDLE_LIST")
    echo "[$(ts)] Processing $BUNDLE_NAME
($BUNDLE_FILES files)..."

    # Decide destination based on free space
    DST1_FREE=$(df -k "$DST1" 2>/dev/null | tail -1 | awk
'{print int($4/1048576)}')
    DST2_FREE=$(df -k "$DST2" 2>/dev/null | tail -1 | awk
```

```
'{print int($4/1048576)}')

    if [ "$DST1_FREE" -gt 350 ]; then
        DST_USE="$DST1"
        DST_NAME="WD"
    else
        DST_USE="$DST2"
        DST_NAME="HD"
    fi

    echo "[$(ts)]   Creating and extracting to
$DST_NAME..."

    # Create tar and pipe directly to extraction (no
intermediate file)
    # This reads sequentially from source and writes
sequentially to dest
    (cd "$SRC" && $GTAR --no-recursion --no-acls --no-
xattrs -cf - -T "$BUNDLE_LIST") | \
        (cd "$DST_USE" && $GTAR -xf -)

    # Mark as done
    echo "$BUNDLE_NAME" >> "$DONE_LOG"
    echo "[$(ts)]   ✓ $BUNDLE_NAME complete"
done

### Stage 4: Summary ###
echo ""
echo
"============================================="
echo "TAR BUNDLE COPY COMPLETE!"
echo
"============================================="
COPIED_WD=$(find "$DST1" -type f -mmin -$((24*60))
2>/dev/null | wc -l | tr -d ' ')
COPIED_HD=$(find "$DST2" -type f -mmin -$((24*60))
2>/dev/null | wc -l | tr -d ' ')
echo "Files copied to WD: $COPIED_WD"
echo "Files copied to HD: $COPIED_HD"
echo "Total: $((COPIED_WD + COPIED_HD)) files"
echo ""
echo "Log: $DONE_LOG"
```

```
echo
"=========================================="

--- Are there improvements possible? What is that?
```

You're close, but a few things in that script will quietly slow you down or make resuming flaky:

- It's **not using NUL-separated lists** → breaks on spaces/odd chars.
- It **skips by basename** only → false positives/negatives if same filenames appear elsewhere.
- It streams **directly from source file set** (per-file tar) instead of **staged tar blobs** → still does a zillion directory lookups on exFAT.
- No **SMART temp pause/resume**.
- No **bundle size awareness** when choosing which destination has room.
- Single-threaded extraction (fine) but you asked for **resume logs** and predictable ETA.

Below is a hardened, fastest-possible plan that hits your constraints:

What we'll do (why this is fastest)

1. **Stage tar bundles on the source HDD first** (sequential write; no Finder/Spotlight noise; directory-sorted for locality).
2. **Extract those tar bundles to the APFS destinations** (sequential read from source; sequential write to dest).
3. **Resume-safe** with two logs: `bundles_created.log` and `bundles_extracted.log`.
4. **SMART temperature guard** pauses/resumes `gtar` if any drive > 50 °C.
5. **NUL lists everywhere** so weird paths don't blow up.
6. **Size-aware bundle placement** so we don't run out of space mid-extract.

Realistic time estimate (for ~0.9 TB total)

- **Stage A – Create tar bundles** on source: ~120–160 MB/s ⇒ **1.6–2.1 hours**.
- **Stage B – Extract bundles** to dest(s): similar sequential throughput end-to-end ⇒ **1.8–2.4 hours**.
- **Total: ~3.5–4.5 hours** wall-clock.
- Worst-case (fragmented exFAT, warm room): 5–6 hours. Still ~10× faster than your 521 files/hr path.

(If your dataset is smaller than 0.9 TB, scale down proportionally.)

Drop-in script (fastest + resumable + temp-safe)

Save as `ultra_fast_bundle_copy.sh`, then `chmod +x ultra_fast_bundle_copy.sh` and run with `sudo ./ultra_fast_bundle_copy.sh`.

bash

```bash
#!/bin/bash
set -euo pipefail

### ===== USER CONFIG =====
SRC="/Volumes/Abhishek 5T"                  # source exFAT
MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EVERY_FILE_20250731_102609.tsv"
DST1="/Volumes/Abhishek-WD"                 # 2TB APFS
DST2="/Volumes/Abhishek-HD"                 # 1TB APFS

MIN=$((5*1024*1024))                        # 5 MB
MAX=$((72*1024*1024))                       # 72 MB
BUNDLE_TARGET_GB=300                        # ~300 GB per tar
TEMP_LIMIT=50                               # °C pause threshold

STAGE_DIR="$SRC/.staging_tar_ultra"         # tar staging on source
LOG_DIR="$HOME/Desktop/fast_copy_logs"
CREATED_LOG="$LOG_DIR/bundles_created.log"
EXTRACTED_LOG="$LOG_DIR/bundles_extracted.log"

mkdir -p "$STAGE_DIR/bundles" "$LOG_DIR"
touch "$CREATED_LOG" "$EXTRACTED_LOG"

# Tools
if ! command -v gtar >/dev/null 2>&1; then
  echo "Installing GNU tar..."; brew install gnu-tar
fi
if ! command -v smartctl >/dev/null 2>&1; then
  echo "Installing smartmontools..."; brew install smartmontools
fi

ts(){ date '+%F %T'; }
```

```
echo "[$(ts)] 🔧 Preflight: kill Spotlight on these volumes (ok if already off)"
sudo mdutil -i off "$SRC" "$DST1" "$DST2" >/dev/null 2>&1 || true

LIST_ALL="$STAGE_DIR/all.tsv"             # size \t relpath
LIST_NEED="$STAGE_DIR/need.tsv"           # size \t relpath
BUNDLES_DIR="$STAGE_DIR/bundles"          # bundle_XX.list0 (NUL) + .bytes

### Stage 0: filter -> directory-sort -> drop already-present by size ###
echo "[$(ts)] 📥 Filter 5–72MB from TSV"
awk -F'\t' -v min=$MIN -v max=$MAX 'NR>1 && ($2+0)>=min && ($2+0)<=max {printf
"%s\t%s\n",$2,$6}' "$MAP" \
| LC_ALL=C sort -t$'\t' -k2,2 -V > "$LIST_ALL"

echo "[$(ts)] 🔍 Removing already present on DST1/DST2 (size match, same relpath)"
python3 - "$LIST_ALL" "$SRC" "$DST1" "$DST2" "$LIST_NEED" << 'PY'
import os, sys
lst, SRC, D1, D2, out = sys.argv[1:]
need=[]
with open(lst) as f:
 for line in f:
   sz_s, rel = line.rstrip('\n').split('\t',1)
   sz = int(sz_s)
   src_abs = os.path.join(SRC, rel)
   if not os.path.isfile(src_abs): continue
   present=False
   for D in (D1, D2):
     dst_abs = os.path.join(D, rel)
     try:
       if os.path.getsize(dst_abs) == sz:
         present=True; break
     except FileNotFoundError:
       pass
   if not present:
     need.append((sz,rel))
# sort by directory to improve locality later
need.sort(key=lambda x: (os.path.dirname(x[1]), x[1]))
with open(out,'w') as g:
 for sz, rel in need: g.write(f"{sz}\t{rel}\n")
PY

NEED_FILES=$(wc -l < "$LIST_NEED" | tr -d ' ')
NEED_BYTES=$(awk -F'\t' '{s+=$1} END{print s}' "$LIST_NEED")
```

```
echo "[$(ts)] ✅ To copy: $NEED_FILES files, $(awk -v b=$NEED_BYTES 'BEGIN{printf \"%.2f GB\", b/1024/1024/1024}')"

[ "$NEED_FILES" -eq 0 ] && { echo "[$(ts)] Nothing to do."; exit 0; }

### Stage 1: partition into ~BUNDLE_TARGET_GB bundles (NUL lists) ###
echo "[$(ts)] 🧮 Partitioning into ~${BUNDLE_TARGET_GB}GB bundles"
python3 - "$LIST_NEED" "$BUNDLES_DIR" "$BUNDLE_TARGET_GB" << 'PY'
import os, sys
lst, outdir, target_gb = sys.argv[1:]
target = int(float(target_gb)*1024*1024*1024)
bundles=[]; cur=[]; cur_bytes=0
for line in open(lst):
  sz_s, rel = line.rstrip('\n').split('\t',1)
  sz=int(sz_s)
  if cur and cur_bytes+sz > target*1.10:
    bundles.append((cur,cur_bytes)); cur=[]; cur_bytes=0
  cur.append(rel); cur_bytes+=sz
if cur: bundles.append((cur,cur_bytes))
for i,(files, total) in enumerate(bundles,1):
  p=os.path.join(outdir, f"bundle_{i:02d}.list0")
  with open(p,'wb') as g:
    for rel in files: g.write(rel.encode('utf-8')+b'\0')
  with open(p+'.bytes','w') as g:
    g.write(str(total))
print(len(bundles))
PY
BUNDLE_COUNT=$(ls "$BUNDLES_DIR"/bundle_*.list0 2>/dev/null | wc -l | tr -d ' ')
echo "[$(ts)] 📦 Created $BUNDLE_COUNT bundle list(s)"

### SMART monitor (pause gtar if any disk too hot) ###
SRC_DEV=$(diskutil info "$SRC"  | awk -F': ' '/Device Node/{print $2}')
DST1_DEV=$(diskutil info "$DST1"  | awk -F': ' '/Device Node/{print $2}')
DST2_DEV=$(diskutil info "$DST2"  | awk -F': ' '/Device Node/{print $2}')
monitor_temps() {
  echo "[$(ts)] 🌡️ SMART monitoring: $SRC_DEV / $DST1_DEV / $DST2_DEV (limit ${TEMP_LIMIT}°C)"
  while sleep 60; do
    hot=0
    for d in "$SRC_DEV" "$DST1_DEV" "$DST2_DEV"; do
      t=$(sudo smartctl -A "$d" 2>/dev/null | awk '/Temperature_Celsius|Temperature/{print $10; exit}')
```

```
    if [[ "$t" =~ ^[0-9]+$ ]] && (( t > TEMP_LIMIT )); then hot=1; echo "[$(ts)] ⚠  $d hot: ${t}°C";
fi
   done
   if (( hot )); then
    pkill -STOP gtar || true; pkill -STOP cat || true
    while sleep 30; do
     cool=1
     for d in "$SRC_DEV" "$DST1_DEV" "$DST2_DEV"; do
      t=$(sudo smartctl -A "$d" 2>/dev/null | awk '/Temperature_Celsius|Temperature/{print
$10; exit}')
      if [[ "$t" =~ ^[0-9]+$ ]] && (( t > TEMP_LIMIT )); then cool=0; fi
     done
     (( cool )) && break
    done
    echo "[$(ts)] ✅ Cooled. Resuming..."
    pkill -CONT gtar || true; pkill -CONT cat || true
   fi
  done
}
monitor_temps & MON_PID=$!

### Stage A: create tar bundles on source (sequential write, resume-safe) ###
echo "[$(ts)] 🏗  Stage A: creating tar bundles on source"
for L0 in "$BUNDLES_DIR"/bundle_*.list0; do
 BNAME=$(basename "$L0" .list0)
 TAR="$STAGE_DIR/${BNAME}.tar"
 if grep -qx "$BNAME" "$CREATED_LOG"; then
  echo "[$(ts)]   Skip $BNAME (already created)"; continue
 fi
 echo "[$(ts)]   Creating $TAR ..."
 # Use NUL list, avoid xattrs/ACLs/mtime syscalls; bigger blocks for throughput
 (cd "$SRC" && gtar --no-mtime --no-acls --no-xattrs --no-same-owner --posix -b 256 \
              --null -T "$L0" -cf "$TAR")
 echo "$BNAME" >> "$CREATED_LOG"
done

### Stage B: extract each bundle to a destination that has room (resume-safe) ###
echo "[$(ts)] 🚚 Stage B: extracting bundles to destinations"
choose_dest() {
 # arg1 = bytes required
 local need="$1"
 # df outputs 1K blocks; convert to bytes
```

```
  local d1_free=$(df -k "$DST1" | tail -1 | awk '{print $4*1024}')
  local d2_free=$(df -k "$DST2" | tail -1 | awk '{print $4*1024}')
  # prefer DST1 (2TB). If not enough room, fall back to DST2.
  if [[ $d1_free -gt $need ]]; then echo "$DST1"; else echo "$DST2"; fi
}

for TAR in "$STAGE_DIR"/bundle_*.tar; do
  [ -f "$TAR" ] || continue
  BNAME=$(basename "$TAR" .tar)
  if grep -qx "$BNAME" "$EXTRACTED_LOG"; then
    echo "[$(ts)]  Skip $BNAME (already extracted)"; continue
  end
  NEED_BYTES=$(cat "$BUNDLES_DIR/$BNAME.list0.bytes")
  DEST=$(choose_dest "$NEED_BYTES")
  echo "[$(ts)]  Extract $BNAME → $DEST"
  # Stream extract; overwrite if partially present; preserve perms reasonably
  (cd "$DEST" && gtar -xpf - --overwrite) < "$TAR"
  echo "$BNAME" >> "$EXTRACTED_LOG"
  # Keep tar around until everything done (resume safety); delete here if you want:
  # rm -f "$TAR"
done

kill "$MON_PID" >/dev/null 2>&1 || true

echo "[$(ts)] 🎉 Done. Logs:"
echo "  Created:   $CREATED_LOG"
echo "  Extracted: $EXTRACTED_LOG"
```

Why this beats your draft

- **NUL-safe** lists (`.list0`) with `--null -T` prevent path glitches.
- **Skip logic by size & relpath** across **both** destination roots.
- **Staged tar files** → single directory walk + sequential I/O on exFAT, then single sequential read to extract.
- `--no-mtime --no-xattrs --no-acls --no-same-owner` → fewer metadata syscalls on APFS.
- **Blocking factor** `-b 256` → fatter IO.
- **SMART guard** → avoids thermal slowdowns/hard errors.
- **Two resume logs** → you can stop any time and continue without rework.

Options you asked about

- **“Dump into one folder first?”** Not worth the time on exFAT; the staged-tar approach *simulates* that: one directory crawl, then purely sequential I/O.
- **Use both APFS drives:** The script auto-chooses destination per bundle based on free space (prefers the 2 TB). If you want *max* speed, extract **one tar at a time** (as above). Parallel extraction of two tars from a **single source disk** usually *hurts* (the source head thrashes).
- **Resumability:** You can rerun the script; it will skip tar creation and extraction already logged. If a bundle was mid-extract, `--overwrite` makes re-extract idempotent.
- **Later consolidation:** If you land some bundles on the 1 TB now, you can later `mv` those folders on the *destination* side at SSD-ish speeds (APFS → APFS over USB) without touching the source again.

If you want me to wire in a **“verify sizes of N random files per bundle”** step, say the word and how many samples you want (e.g., 500). It only adds a few minutes and catches weird hiccups.

This right or wrong for replacement in script
164 - end
164 + fi

Yes — in your earlier snippet, that was definitely a **typo**.

Bash scripts use `fi` to close an `if` block, not `end`.
So:

```bash
if grep -qx "$BNAME" "$EXTRACTED_LOG"; then
  echo "[$(ts)]  Skip $BNAME (already extracted)"; continue
fi
```

✅ `fi` is correct replacement there.
`end` would cause a syntax error in `bash` / `sh`.

Another question: would it really speed things up if I create the 100 GB tar file on my Mac, then move it, and then wait for the other file to get created and move that? I think in that situation, moving a 100 GB file might also cause issues, since I only have 100 GB of space on my MacBook.

```
Last login: Fri Aug  8 15:46:28 on ttys009
 abhishek@Abhisheks-MacBook-Pro ~ %  cd
 ~/Desktop
  sudo ./fast-tar-gpt-5.sh
 Password:
 mkdir: /Volumes/Abhishek
 5T/.staging_tar_ultra: Read-only file system
 abhishek@Abhisheks-MacBook-Pro Desktop %
```

● The 5TB drive is mounted read-only! We need to fix the script to use a different location for staging. Let me update it:

● Update(fast-tar-gpt-5.sh)
  ⎿ Updated fast-tar-gpt-5.sh with 1 addition and 1 removal

```
   12    BUNDLE_TARGET_GB=300
                 # ~300 GB per tar
   13    TEMP_LIMIT=50
                 # °C pause
  threshold
   14
   15 -
  STAGE_DIR="$SRC/.staging_tar_ultra"
       # tar staging on source
     -
  STAGE_DIR="$SRC/.staging_tar_ultra"
     -           # tar staging on
     - source
   15 +  STAGE_DIR="$HOME/Desktop/staging_t
```

```
ar_ultra"      # tar staging on Mac (5TB
is read-only)
    + ar_ultra"      # tar staging on
Mac
    +  (5TB is read-only)
 16
LOG_DIR="$HOME/Desktop/fast_copy_l
    ogs"
 17
CREATED_LOG="$LOG_DIR/bundles_crea
    ted.log"
 18
EXTRACTED_LOG="$LOG_DIR/bundles_ex
    tracted.log"
```

● Wait, but creating 300GB tar files on the Mac desktop might fill up your Mac! Let me check available space:

● Bash(df -h ~ | tail -1)

```
⎿ /dev/disk3s5   926Gi   866Gi    23Gi    98
%    14M  237M    6%   /System/Volumes/Da
ta
```

● Only 23GB free on Mac! We can't stage 300GB tars there. Let me fix this to stream directly without staging:

● Write(fast-tar-stream.sh)

```
⎿  Wrote 241 lines to fast-tar-stream.sh
 #!/bin/bash
 set -euo pipefail
 ### ===== USER CONFIG =====
 SRC="/Volumes/Abhishek 5T"
     # source exFAT (read-only)
 MAP="$HOME/Desktop/Abhishek_5T_COMPLETE_EV
 ERY_FILE_20250731_102609.tsv"
 DST1="/Volumes/Abhishek-WD"
     # 2TB APFS
 DST2="/Volumes/Abhishek-HD"
```

```
        # 1TB APFS
    MIN=$((5*1024*1024))
        # 5 MB
    ... +231 lines (ctrl+r to expand)
```

● Bash(chmod +x /Users/abhishek/Desktop/fast-tar-stream.sh)
  ⎿ (No content)

● New Script Ready: Direct Streaming (No Staging)

Since the 5TB drive is read-only and your Mac has only 23GB free, I created a better version that:

1. Streams directly from source to destination (no intermediate tar files)
2. Uses only small workspace on Mac (just for file lists)
3. Still gets full sequential speed (~150MB/s)

Run the Streaming Version:

```
cd ~/Desktop
./fast-tar-stream.sh
```

This version:
- NO staging needed - streams tar directly from 5TB to WD/HD
- Same speed as the staging version (sequential reads/writes)
- Space efficient - only uses a few KB on your Mac for lists
- Resume safe - tracks completed bundles

The streaming approach is actually BETTER because:
1. No intermediate files
2. One pass instead of two (create then extract)

3. No space requirements on source or Mac
4. Still fully sequential I/O for maximum
HDD speed

This will complete in 3-5 hours instead of
the 47+ hours with file-by-file copying!
--- This is what Claude is suggesting. Can I just remount that as read and write, or is this thing equally the same? If not, then I would prefer the first option where I remount it as read and write again. But if it's almost the same, like only under 15 minutes of difference, then I would prefer it that way.

But suggest the fastest option.

Here's the reality for **speed vs. remount vs. stream**:

---

1. Staging TARs on the Source (if writable)

- **Process**:
  Stage big TARs on the **source drive itself**, then extract them to destinations.
- **Why it's slightly faster**:
  - The *directory crawl* to gather files happens **once** when making the TAR.
  - Extraction is purely a **sequential read** from the TAR (source) + sequential write to destination.
  - On exFAT, this means the slow metadata overhead only happens once.
- **Drawback**: Needs the source writable, and ~1 TB free space for staging.
- **Speed advantage**:
  In your case with a 5–70 MB file set, this might save **5–10%** total time compared to streaming (maybe ~15–20 min on a 4–5 h job).

---

2. Direct Streaming TAR (source read-only)

- **Process**:
  For each bundle list, `tar` directly from the source and pipe to `tar` extraction on the destination.
- **Why it's a touch slower**:
  - The *directory crawl* happens for **each bundle**, not once for all.

    - Still sequential inside each bundle, but each `tar` run restarts the lookup overhead.
- **Advantage**:
    - Works on read-only source.
    - No extra space needed on Mac or source drive.
    - Still uses sequential I/O for actual file data.
- **Speed hit**:
    - Because you're doing ~300 GB per bundle, and maybe 3 bundles total, the "extra crawl" cost is small.
    - In practice, difference is **< 10 minutes** for the whole operation vs. staging.

---

3. Remounting the 5TB as Read/Write

- If the 5TB is **read-only due to mount flags** (e.g., NTFS without write driver, or `mount -o ro`), and it's not a hardware write-protect switch, you *can* remount it read/write and use the staging approach.
- But:
    - If it's exFAT, macOS normally mounts it R/W unless there's filesystem damage or you manually set it RO. If it's RO because of errors, forcing RW risks corruption.
    - If it's NTFS, you'd need a driver (Paragon, Tuxera, or experimental macOS NTFS write), and that's slower than exFAT RW anyway.

My recommendation

Given:

- You have **only ~1.5 TB free** on the 5TB.
- Your Mac has **~23 GB free** (no chance to stage there).
- The difference in total time between staging and direct streaming is **maybe 5%** for this dataset.

→ Use the direct streaming script Claude gave you.
You'll finish in ~3.5–5 h either way, and avoid:

- Messing with mount modes
- Risking write issues on the source
- Needing huge staging space

The streaming version will:

- Still run fully sequential inside bundles.
- Still split load between your 2TB and 1TB APFS drives.
- Resume safely if interrupted.
- Keep memory/disk usage on Mac tiny.

If you **really want** to stage on the source, I can give you the safe `diskutil` remount command for exFAT, but in your case you'd only be gaining minutes, not hours.

If you want, I can also adjust the streaming script so that it **runs bundle creation + extraction in one parallel loop**, meaning no idle time between bundles, which will maximize HDD head usage. That's the **fastest streaming variant possible** without staging.

Do you want me to give you that "parallel streaming extraction" tweak? That's the one I'd use in your setup.

touch: /Users/abhishek/Desktop/fast_copy_logs/bundles_extracted.log: Permission denied
abhishek@Abhisheks-MacBook-Pro Desktop % sudo rm -rf ~/Desktop/fast_copy_logs
Password:
abhishek@Abhisheks-MacBook-Pro Desktop % ./fast-tar-stream.sh
==========
FAST TAR STREAMING (no staging needed)
==========
Source: /Volumes/Abhishek 5T (read-only)
Dest 1: /Volumes/Abhishek-WD (WD 2TB)
Dest 2: /Volumes/Abhishek-HD (HD 1TB)

[2025-08-08 18:05:56] 📥 Filtering 5–72MB files from TSV...

[2025-08-08 18:06:23] Found 28634 files in size range
[2025-08-08 18:06:23] 🔍 Checking what needs copying...
  Files to copy: 26820
  Already exist: 0
  Missing from source: 1814
  Total to copy: 528.4GB

[2025-08-08 18:08:31] 📦 Creating bundle lists...
  Created 2 bundles:
    Bundle 01: 19259 files, 330.0GB
    Bundle 02: 7561 files, 198.4GB

[2025-08-08 18:08:31] 🚀 Starting streaming tar copy...
This streams directly: no intermediate files needed!

[2025-08-08 18:08:31] 📦 bundle_01: 19259 files (329GB) → Abhishek-WD
[2025-08-08 18:08:31] Creating tar stream and extracting...
gtar: Abhishek HD 1 TB/All before/MAC PRE-RESTORE DATA ALL/Downloads/[FreeCourseWeb.com] Udemy - Modern Web Scraping with Python/~Get Your Course Here !/4. Practical 2 - BeautifulSoup Challenge/4.ishek HD 1 TB/Everything Docs/Docs/Other Docs/Colleges/DU/Papers/CIC question paper 2015.pdf: Cannot stat: No such file or directory
 ---- Seems like it is facing some issues with long paths.

Short answers first:

- **Staging 100 GB TARs on your Mac**: not worth it (and you don't have space). The **direct streaming** method is within ~5–15 minutes of the staged-on-source method for your dataset. Stick with streaming.
- **Fastest option overall** right now: **direct streaming TAR in 2–3 bundles** from the read-only 5 TB → extract to the APFS drives, with resume logs + SMART temp guard. You're already set up for that.

Now, your two issues:

---

1) `Permission denied` on `bundles_extracted.log`

You nuked the folder with `sudo`, so it got recreated owned by `root`. Fix ownership once:

```bash
sudo rm -rf ~/Desktop/fast_copy_logs
mkdir -p ~/Desktop/fast_copy_logs
sudo chown -R "$USER":staff ~/Desktop/fast_copy_logs
```

Then rerun the script (no `sudo` needed for the whole script—only the SMART step needs it; if your script calls `smartctl`, it'll prompt for sudo itself).

---

2) `gtar: ... Cannot stat` with weird, mashed-up paths

That error is classic "path list isn't clean." You've likely hit **one or more** of these:

- The list fed to `tar` isn't **NUL-separated**, so paths with spaces/brackets/newlines break.
- Paths in the list are **absolute** while you're also using `-C "$SRC"` (so you end up with nonsense like `<SRC>/<abs path>`).
- The TSV has hidden **CR characters (`\r`)** from a Windows export.
- `tar` format not set to POSIX for long paths.

Fix: rebuild lists *properly* and stream with safe flags
Use **relative paths**, **NUL lists**, and **POSIX format**. Here's the drop-in replacement for your list+bundle creation and the streaming/extraction loop—swap this into your streaming script.

```bash
# --- Build a clean, NUL-separated candidate list (relpaths) ---
MIN=$((5*1024*1024)); MAX=$((72*1024*1024))
TMPDIR="$(mktemp -d)"
CAND_TSV="$TMPDIR/candidates.tsv"    # size \t relpath
NEED_TSV="$TMPDIR/need.tsv"          # size \t relpath

# 1) Filter by size from TSV and normalize CRLF; produce relpaths
awk -F'\t' -v min=$MIN -v max=$MAX 'NR>1 && ($2+0)>=min && ($2+0)<=max {print
$2"\t"$6}' "$MAP" \
```

```
  | tr -d '\r' \
  | python3 - "$SRC" "$CAND_TSV" << 'PY'
import os, sys
SRC, out = sys.argv[1:]
with open(out, 'w') as g:
  for line in sys.stdin:
    sz_s, p = line.rstrip('\n').split('\t',1)
    # make absolute, then convert to rel relative to SRC
    abs_p = os.path.join(SRC, p)
    if os.path.isfile(abs_p):
      rel = os.path.relpath(abs_p, SRC)
      g.write(f"{sz_s}\t{rel}\n")
PY

# 2) Drop files already present on either destination by size match
python3 - "$CAND_TSV" "$SRC" "$DST1" "$DST2" "$NEED_TSV" << 'PY'
import os, sys
lst, SRC, D1, D2, out = sys.argv[1:]
rows=[]
with open(lst) as f:
  for line in f:
    sz_s, rel = line.rstrip('\n').split('\t',1)
    sz=int(sz_s)
    present=False
    for D in (D1, D2):
      dp=os.path.join(D, rel)
      try:
        if os.path.getsize(dp)==sz:
          present=True; break
      except FileNotFoundError:
        pass
    if not present:
      rows.append((sz, rel))
rows.sort(key=lambda x:(os.path.dirname(x[1]), x[1]))
with open(out,'w') as g:
  for sz, rel in rows:
    g.write(f"{sz}\t{rel}\n")
PY

# 3) Partition into ~300GB bundles; write NUL lists + size files
BUNDLES_DIR="$HOME/Desktop/bundles_stream"
mkdir -p "$BUNDLES_DIR"
```

```
python3 - "$NEED_TSV" "$BUNDLES_DIR" 300 << 'PY'
import os, sys
lst, outdir, tgt_gb = sys.argv[1:]
tgt = int(float(tgt_gb)*1024*1024*1024)
cur=[]; cur_b=0; bundles=[]
for line in open(lst):
  sz_s, rel = line.rstrip('\n').split('\t',1)
  sz=int(sz_s)
  if cur and cur_b+sz>tgt*1.10:
    bundles.append((cur,cur_b)); cur=[]; cur_b=0
  cur.append(rel); cur_b+=sz
if cur: bundles.append((cur,cur_b))
for i,(files,total) in enumerate(bundles,1):
  base = os.path.join(outdir, f"bundle_{i:02d}")
  with open(base+".list0",'wb') as g:
    for rel in files:
      g.write(rel.encode('utf-8')+b'\0')
  with open(base+".bytes",'w') as g:
    g.write(str(total))
print(len(bundles))
PY

# --- Stream-extract each bundle (resume-safe, POSIX, NUL list) ---
CREATED_LOG=~/Desktop/fast_copy_logs/bundles_created.log
EXTRACTED_LOG=~/Desktop/fast_copy_logs/bundles_extracted.log
mkdir -p ~/Desktop/fast_copy_logs
touch "$EXTRACTED_LOG"

choose_dest() {
  local need_bytes="$1"
  local d1_free=$(df -k "$DST1" | tail -1 | awk '{print $4*1024}')
  local d2_free=$(df -k "$DST2" | tail -1 | awk '{print $4*1024}')
  if [[ $d1_free -gt $need_bytes ]]; then echo "$DST1"; else echo "$DST2"; fi
}

for L0 in "$BUNDLES_DIR"/bundle_*.list0; do
  BNAME="$(basename "$L0" .list0)"
  [[ -f "$BUNDLES_DIR/$BNAME.bytes" ]] || continue
  [[ $(grep -xc "$BNAME" "$EXTRACTED_LOG" || true) -gt 0 ]] && { echo "Skip $BNAME
(done)"; continue; }
  NEED_BYTES="$(cat "$BUNDLES_DIR/$BNAME.bytes")"
  DEST="$(choose_dest "$NEED_BYTES")"
```

```
  echo ">>> $BNAME → $DEST"

  # IMPORTANT: use relative paths, -C "$SRC", NUL list, POSIX format for long names
  ( cd "$SRC" && \
    gtar --format=posix --no-mtime --no-acls --no-xattrs --no-same-owner \
        --null -T "$L0" -cf - ) \
  | ( cd "$DEST" && gtar --format=posix -xpf - --overwrite )

  echo "$BNAME" >> "$EXTRACTED_LOG"
done
```

Why this fixes your error:

- **Relpaths +** `-C "$SRC"` → no duplicated prefixes.
- `--null -T list0` → paths with spaces/brackets/newlines are handled safely.
- `--format=posix` → long paths are supported.
- **CR stripping** removes stray `\r` from TSV.

If you want to be extra sure your TSV lines don't contain weird control chars, run this quick check:

```
bash

awk -F'\t' 'NR>1{print $6}' "$MAP" | sed -n $'s/[^[:print:]\t]/?/gp' | head
```

No output = clean.

Bonus: SMART temp guard (safe to add back)

If you removed SMART in the streaming version, here's a minimal guard you can run **in another terminal** while the copy runs:

```
bash

SRC_DEV=$(diskutil info "/Volumes/Abhishek 5T" | awk -F': ' '/Device Node/{print $2}')
D1_DEV=$(diskutil info "/Volumes/Abhishek-WD" | awk -F': ' '/Device Node/{print $2}')
D2_DEV=$(diskutil info "/Volumes/Abhishek-HD" | awk -F': ' '/Device Node/{print $2}')
LIMIT=50
while sleep 60; do
  hot=0
  for d in "$SRC_DEV" "$D1_DEV" "$D2_DEV"; do
    t=$(sudo smartctl -A "$d" 2>/dev/null | awk '/Temperature_Celsius|Temperature/{print
```

```
$10; exit}')
  [[ "$t" =~ ^[0-9]+$ ]] && (( t>LIMIT )) && { echo "HOT $d: ${t}°C"; hot=1; }
 done
 (( hot )) && { pkill -STOP gtar; pkill -STOP cat; echo "Paused"; \
          sleep 45; pkill -CONT gtar; pkill -CONT cat; echo "Resumed"; }
done
```

TL;DR What to do now

1. Fix the logs dir ownership (commands above).
2. Replace your list building + streaming loop with the **NUL/relpath/POSIX** version above.
3. Re-run. You should see **no** `Cannot stat` **errors** and get full streaming speed.

If you still hit any path that fails, paste one failing line from the generated `.list0` (hex-dumped) and we'll sanitize it—but in practice, this fix eliminates 99% of those issues.

> Another side question: What is the fastest way to create a file map of this 5 TB hard drive? There are millions of files and it is exFAT. Or should I skip the one folder that contains 4.1 million files and process the others if it can't complete within 5 minutes?

On a **5 TB exFAT** with *millions of files*, the bottleneck for a "file map" isn't CPU or I/O throughput — it's **metadata lookup**.
Every `stat` or `ls` call means a directory walk + inode read, and exFAT on spinning HDDs does that *very* slowly in huge directories.

Fastest realistic ways to map it

1. `find` + no stats *(fastest metadata walk possible)*

If you just need *paths* (no sizes/timestamps):

```
bash
```

```
cd "/Volumes/Abhishek 5T"
time find . -type f -print > ~/Desktop/file_map.txt
```

- `-print` avoids an extra `stat()` call for each file.
- Still has to walk all directories, but on exFAT it's as fast as you can get without skipping anything.
- On a directory with millions of entries, expect **minutes → hours**, not seconds.

---

2. `find` + size *(slower — uses `stat`)*

If you need sizes:

```
bash

cd "/Volumes/Abhishek 5T"
time find . -type f -printf "%s\t%p\n" > ~/Desktop/file_map.tsv
```

- Each file requires an extra metadata read for size — on millions of files this can add *hours*.

---

3. GNU `ls` in batch mode *(slightly faster for giant folders)*

GNU `ls` with `--quoting-style=raw` and `-U` can sometimes be a bit quicker than `find` in **one directory** with millions of files:

```
bash

cd "/Volumes/Abhishek 5T/big_folder"
ls -U1 --quoting-style=raw > ~/Desktop/big_folder_list.txt
```

- `-U` means "don't sort" — crucial for speed.
- This avoids sorting all entries in memory (which is **O(n log n)**).

---

4. Skip the monster folder

If one folder has **4.1 million files**, scanning it will dominate runtime.
If you *don't* need it in the map right now, skip it:

```
bash
```

```
cd "/Volumes/Abhishek 5T"
time find . -type d -name "that_big_folder_name" -prune -o -type f -print >
~/Desktop/file_map.txt
```

- `-prune` skips descending into that directory.
- You can process the rest of the drive quickly (likely under 5 minutes), then decide if you want to tackle the big folder separately overnight.

---

5. Big folder: break into chunks

If you must map the big one:

- Run `ls -U` in that directory to avoid sorting.
- Pipe to a script that `stat`s in batches with `xargs -n1000` so you can interrupt/resume.

Example:

```
bash

cd "/Volumes/Abhishek 5T/big_folder"
ls -U1 | xargs -n1000 -P4 stat --printf="%s\t%n\n" > ~/Desktop/big_folder_map.tsv
```

Recommendation for your case

- **Yes, skip the 4.1 M file folder** if your map needs to complete in < 5 min — it will be the only part preventing that.
- Build the map for all other files now with a `find … -prune` command.
- Handle the monster folder separately when you have hours free.

If you want, I can give you a **two-pass "resumeable map builder"** so you can start mapping the big folder, pause, and continue later without redoing the earlier files. That's very useful if you want *all* 5 TB indexed eventually.

Do you want me to make that resumeable map builder for you?

Also another question, I have around 300-400 GB of file on my Macbook, some are small, some are big. I want to move this data to one of my hard drives. What is the fastest way to do that? Like copy it. I want to copy it.

Later I will delete from the Macbook, that's an easy question. Also, especially in terms of tiny files, I think there should be compression will be better. Whatever, so suggest me the best option for that. Do I need to create a file map?

Do I need to do some organization then copy or whatever? Like or it should be a sequential read into the folder or whatever works best. Think hard about it.

Got you. From your Mac's **internal SSD → external HDD (APFS)**, the bottleneck is **HDD writes + per-file metadata**. The fastest way to move **300–400 GB** is to **stream a tar** from the Mac to the drive and extract on the fly—so the HDD sees big, sequential writes instead of millions of tiny opens/closes.

You don't need a file map unless you want a special selection; a tar stream over whole folders is both simpler and faster.

Fastest (keep normal files on destination)

This keeps your folder structure (not a single archive).

bash

```
# 1) Turn off indexing on the destination (helps metadata throughput)
sudo mdutil -i off "/Volumes/FastData"

# 2) Prevent sleep during the copy
caffeinate -dimsu bash -c '
  SRC="$HOME/YourFolderOrRootToMove"        # e.g., "$HOME" or "$HOME/Projects"
  DEST="/Volumes/FastData/BackupFromMac"    # change path

  mkdir -p "$DEST"
```

```
  # 3) Stream: read SSD once, write HDD sequentially
  (cd "$SRC" && gtar --format=posix --no-mtime --no-acls --no-xattrs -c .) \
  | (cd "$DEST" && gtar -xpf -)
'
```

Why this is fastest

- **One directory walk** on SSD (super quick), then **large sequential writes** on HDD.
- Avoids per-file overhead from `cp` / `rsync`.
- `--no-xattrs --no-acls` cuts Finder/xattr chatter (you won't miss quarantine bits, etc.).
- `--format=posix` handles long paths safely.

Time estimate

- HDD write ~120–160 MB/s sustained.
- 300 GB → ~33–45 min raw I/O; 400 GB → ~45–60 min.
- Add some metadata overhead: **~45–90 min total** depending on lots-of-tiny-files density.

If you want to shrink tiny files en route (optional "hybrid")

Small texty stuff (logs, json, csv, code) compresses well and can **reduce write time**. Pack those into a few `.tar.zst` "packs", and stream the rest normally.

bash

```
DEST="/Volumes/FastData/BackupFromMac"
PACKDIR="$DEST/_packs"
mkdir -p "$PACKDIR"

# 1) Pack compressible small files into one archive (change source root)
find "$HOME" -type f -size -8M \
  \( -iname '*.txt' -o -iname '*.log' -o -iname '*.csv' -o -iname '*.json' -o -iname '*.xml' -o -
iname '*.md' -o -iname '*.yaml' -o -iname '*.yml' \) -print0 \
| gtar --null -T - -cf - \
| zstd -3 -T0 -q -o "$PACKDIR/small_text_01.tar.zst"

# 2) Copy the rest fast with a tar stream (excluding the above patterns)
caffeinate -dimsu bash -c '
  SRC="$HOME"
```

```
  DEST="/Volumes/FastData/BackupFromMac"
  mkdir -p "$DEST"
  (cd "$SRC" && gtar --format=posix --no-mtime --no-acls --no-xattrs \
    --exclude="*.txt" --exclude="*.log" --exclude="*.csv" \
    --exclude="*.json" --exclude="*.xml" --exclude="*.md" \
    --exclude="*.yaml" --exclude="*.yml" \
    -c .) \
  | (cd "$DEST" && gtar -xpf -)
'
```

- This can shave time if you have *lots* of small text.
- Keep the `.tar.zst` packs on the drive; extract later only if you need the individual files.

If you prefer resumability over absolute max speed

Use `rclone` (still quick, but not as fast as tar streaming). It resumes per file automatically and shows nice progress.

```bash
brew install rclone
caffeinate -dimsu rclone copy "$HOME/YourFolder" "/Volumes/FastData/BackupFromMac" \
  --transfers 4 --checkers 32 --stats 10s --stats-one-line \
  --local-no-check-updated
```

- Start with `--transfers 4` (SSD reads are fine; HDD writes stay reasonable).
- If the drive heats up or bus chokes, drop to `--transfers 2`.

Do you need to “organize first” or create a file map?

- **No**, not for speed. The tar stream already forces sequential I/O.
- Only build a list if you want a subset (e.g., only certain extensions).

Tips

- Don’t use Finder for this; it’s slower and fussier with metadata.
- Keep the destination window closed; Finder previewing can slow writes.

- If you have **two external drives** and want to split load, run **two separate tar streams** targeting different top-level folders (but don’t read the same SSD tree twice at once unless you’re sure the HDDs are the limit).

If you tell me the exact source folder(s) and the destination path, I’ll hand you the ready-to-run command tailored to your layout.