Abhishek R-P - Stream processing large archive

das_niftyopt_backup.7z.010 --- There are 9 other parts of it. All are around 10 GB each. My question is about the Nifty Options Data file. The other ones are the same.

I want to run a backtest on these files, but together they are about 100 GB. When uncompressed, they were around 600 GB. I don't have 600 GB of space available on my MacBook. I barely have 100 GB.

What is the best possible way for me to go through this?

Got it — your `.7z.001` through `.7z.010` parts together make a **multi-part archive** of a ~600 GB uncompressed dataset, but you only have ~100 GB free.
This means **you cannot simply extract everything** — you need a strategy that lets you read/process the data directly from the compressed parts without fully inflating them to disk.

Here's the best-possible approach options for your situation:

1️⃣ Stream-Process Without Full Extraction (Best Option)

You can **pipe the decompressed output directly into your processing/backtest script** without writing the whole dataset to disk.

How it works

- `7z` or `p7zip` can stream decompression to stdout.
- Your code can read from that stream line-by-line, chunk-by-chunk, or even filter before storing.
- You avoid storing the entire 600 GB; only the parts you're processing at a time exist in memory/disk.

Example

```bash
```

```
7z x -so das_niftyopt_backup.7z.001 | python process_data.py
```

- `-so` = send output to stdout instead of creating files.
- Your `process_data.py` reads from `stdin` and processes immediately.
- Works with CSV/JSON/Parquet inside the archive.
- If the `.7z` contains multiple files, you can target a specific one with `7z x -so archive.7z file_name`.

2️⃣ Mount the Archive as a Virtual Filesystem

Use avfs ↗ **or** ratarmount ↗ to mount `.7z` as a folder.

- Lets you **browse/extract individual files** without decompressing the whole archive.
- Useful if you want to grab only certain date ranges or symbols from the Nifty data.

Example with `ratarmount`:

```bash
pip install ratarmount
ratarmount das_niftyopt_backup.7z.001 /mnt/nifty
# Now /mnt/nifty acts like a folder
```

You can then read only the files you need.

3️⃣ Extract in Batches (Partial Extraction)

If the archive contains daily or monthly files:

- Extract **one chunk at a time**, process, then delete it.
- This is the safest for extremely limited disk space, but slower because of repeated decompressions.

Example:

```bash
7z x das_niftyopt_backup.7z.001 -o/tmp/output specific_file_202301.csv
python process_data.py /tmp/output/specific_file_202301.csv
```

```
rm /tmp/output/specific_file_202301.csv
```

4️⃣ Use an External Drive for Temporary Space

If you have a **fast external SSD**, you can:

- Extract the whole thing to the external drive.
- Process locally without worrying about disk limits.
- This will be much faster than network streaming if you plan multiple passes through the data.

5️⃣ Convert Once to a Space-Efficient Format

If you can **borrow space once** (external SSD, cloud VM), you can:

1. Extract fully.
2. Convert to **columnar compressed format** like Parquet/Zstandard.
3. Reduce 600 GB → potentially <100 GB.
4. Copy back to your MacBook for repeated backtests.

💡 **My Recommendation for You Now** (since you have only 100 GB free):

- **If your backtest reads sequentially** → Use **Option 1 streaming** (`7z -so | python`), no extraction needed.
- **If your backtest needs random file access** → Use **Option 2 ratarmount** to mount and read only what's needed.

If you want, I can give you a **ready-to-run Python backtest loader** that reads your `.7z` parts **on the fly** without extracting them, so you never exceed your 100 GB limit. That way you can directly run the backtest on the compressed dataset.

Do you want me to prepare that?

I have an external hard drive that I can use, and it has about 2 TB of space available. My question is, will it be fast enough to read through that when I run the backtest? What I usually do during the backtest is load all the data using NumPy, JIT, and TA-Lib libraries into an array, and then run the backtest on that.

Short answer: **yes—if it's a fast SSD over USB-C/Thunderbolt.** If it's a spinning HDD, it'll *work* but feel sluggish.

Here's how to decide + what to change for best performance.

What matters

- **Drive type**
  - **HDD (spinning):** ~120–180 MB/s sequential, terrible random I/O. 600 GB full read ≈ **55–80 min**.
  - **SATA SSD (USB 3.1/3.2 Gen1):** ~450–550 MB/s. 600 GB read ≈ **18–22 min**.
  - **NVMe SSD (USB 3.2 Gen2/Gen2x2):** ~900–1,800 MB/s. 600 GB read ≈ **6–12 min**.
  - **NVMe SSD (Thunderbolt 3/4):** ~2,000–2,800 MB/s. 600 GB read ≈ **4–6 min**.
- **Interface/cable**: USB-C ≠ always fast; you want **USB 3.2 Gen2 (10 Gbps) or Thunderbolt**. Avoid hubs/dongles that downshift to 5 Gbps.
- **Your access pattern**: TA-Lib indicators are mostly **sequential/windowed** → you benefit most from **sequential throughput** and not so much from random IOPS.

Practical recommendations (what I'd do)

1. **Use a fast external SSD** (NVMe in a USB-C Gen2 or Thunderbolt enclosure). If your 2 TB drive is an HDD, consider borrowing/buying a USB-C NVMe stick—it'll save you hours every run.
2. **Keep everything on the external SSD**
   - Copy the `.7z.001–.010` to the SSD and **extract there**.
   - Decompression (LZMA) is **CPU-bound**; on Apple Silicon you'll often see ~300–800 MB/s with multi-threading, so the SSD won't be your bottleneck.
   - Use `7z x -mmt=on` to enable multithreaded decompression.
3. **Don't load the entire dataset into RAM**
   - Use **memory-mapped arrays** (`numpy.memmap`) so your code can address 600 GB like an array while macOS handles paging from the SSD.

- Or convert once to **columnar Parquet** (Snappy/ZSTD) partitioned by date/symbol—then read the exact slices you need.

4. **Stream or batch**
   - If your backtest is pass-based (single sequential sweep), stream day-by-day/month-by-month into arrays, compute, discard.
   - If you need random access across the full time span, **memmap** or **Parquet+Polars/PyArrow** beats giant in-RAM arrays.

Minimal setup snippets

Faster, multi-threaded extract (on the external SSD)

```bash
# assume /Volumes/FastSSD
cd /Volumes/FastSSD
7z x -mmt=on /Volumes/FastSSD/das_niftyopt_backup.7z.001
```

Memory-map for TA-Lib / NumPy

```python
import numpy as np
# Suppose you saved a giant float32 matrix to .npy once
arr = np.memmap("/Volumes/FastSSD/nifty_ticks.npy", dtype='float32', mode='r', shape=(N,
C))
# Use arr like a normal ndarray; OS pages from SSD
```

Columnar once, fast reads forever (recommended if you'll iterate often)

- Convert raw CSV → **Parquet partitioned by date/symbol** (one-time job on the SSD).
- Then:

```python
import polars as pl
df = pl.scan_parquet("/Volumes/FastSSD/nifty_parquet/date=2023-01/*.parquet") \
      .select(["ts","symbol","strike","price","oi"]) \
      .collect()
# super fast columnar reads, filter/predicate pushdown
```

Filesystem & macOS tips

- **Format the SSD as APFS** (or HFS+) on macOS; avoid exFAT (slower, flaky metadata).
- Disable Spotlight indexing on that volume for a tiny boost:
    - System Settings → Siri & Spotlight → Privacy → add the drive.
- Ensure the enclosure/cable supports the advertised speed (Gen2 10 Gbps or TB3/4).

Benchmark your actual setup (takes 1–2 minutes)

```bash
# quick sequential read test on a big file
time dd if=/Volumes/FastSSD/bigfile of=/dev/null bs=64m count=200
# or install fio for a more realistic profile:
brew install fio
fio --name=read --rw=read --bs=1m --size=10g --filename=/Volumes/FastSSD/testfile --
iodepth=32
```

Bottom line

- **HDD**: will work, but expect ~1 hour just to scan 600 GB each pass.
- **SATA SSD**: totally fine; ~20 minutes per full-dataset pass.
- **NVMe over USB-C Gen2/Thunderbolt**: ideal; **single-digit minutes** per pass.
- Combine that with **memmap/Parquet** and you won't need 600 GB free space on the MacBook.

If you tell me what exact drive/enclosure you have (HDD vs SSD, interface), I'll sanity-check the expected speeds and suggest the best loading approach (memmap vs Parquet) for your current backtest code.

Okay, initially it was in a MariaDB database within SQL tables. There are thousands of SQL tables. If I convert that to Parquet, how much space is it going to take, roughly? Also, I would like to run tests on a month-by-month basis or for 6 months at a time. I don't currently have an SSD. I would have to buy one just for this, and I don't want to. What I can do is use 100 GB of space on my Mac and use that. I can temporarily put some files here and there, etc.

Short version: converting to Parquet will *massively* shrink it, and you can work **month-by-month** within your 100 GB limit—even using a slow external HDD—if you pipeline it right.

How much space will Parquet take?

Rough ballparks from real-world market/tick datasets:

- **CSV/raw rows → Parquet (ZSTD/Snappy):** ~**3×–10× smaller** (often 10–30% of CSV size).
- **MySQL/MariaDB InnoDB files → Parquet:** typically **2×–5× smaller** (20–50% of DB size) because Parquet uses columnar encodings, dictionary/RLE, delta for timestamps, etc., while InnoDB stores rows + indexes + page overhead.

Given your “~600 GB uncompressed” figure (likely DB files or CSV dumps):

- Conservative: **~200–300 GB** Parquet total (33–50%).
- Optimistic for numeric, repetitive columns (options data): **~90–180 GB** (15–30%).

So: **expect ~100–250 GB** after Parquet depending on schema/indices.
Either way: totally viable if you process **by month** and keep only the months you need locally.

Work within 100 GB: month-by-month pipeline

You don’t need to buy an SSD. Use your 100 GB internal + the HDD as a **staging/long-term store**, and keep your *active month* on the Mac’s internal disk for faster compute.

Strategy

1. **Stage + convert once → Parquet partitioned by month**
   - Keep the **Parquet master copy on the external drive** (HDD).
   - Convert **one month at a time** so you never exceed ~50–80 GB transient usage.
   - After converting a month, **delete the raw for that month**.
2. **Backtest in slices**
   - For each run, copy (or directly scan) **1 month** (or **6 months**) of Parquet to your internal disk (or read directly if OK with HDD speed).
   - Load into NumPy (or use Polars/Arrow to pre-aggregate) and run.

Two practical conversion flows

A) If you have to restore from MariaDB `.ibd/.frm` (or .SQL dump)

- Put MariaDB **datadir on the external HDD** so you don't fill internal disk.
- Restore DB.
- For each table/month:
  - Use `SELECT … WHERE date BETWEEN … INTO OUTFILE 'month.csv'` (or `mysqldump --where`), preferably **compressed** (e.g., pipe to `gzip`).
  - Convert `month.csv(.gz)` → **Parquet (ZSTD)** and **partition by month**.
  - Delete the CSV.

B) If you already have CSVs (or can stream from `.7z`)

- Extract or **stream** one month's worth at a time.
- Convert straight to Parquet; delete the CSV immediately.

Parquet settings that help

- **Compression:** ZSTD level 3–5 (good ratio, fast).
- **Row group size:** 128 MB (default) is fine.
- **Dictionary encoding:** on (default).
- **Partitioning:** `year=YYYY/month=MM/` folders.

Minimal code (DuckDB = simplest, fastest converter)

DuckDB can read CSV (even `.gz`) and **write partitioned Parquet** in one step.

```bash
brew install duckdb
```

```sql
-- duckdb -c "<SQL here>"  OR launch duckdb shell and run:

-- Example: one huge CSV of many days, filtered to Jan 2023
COPY (
  SELECT *
  FROM read_csv('/Volumes/ExtHDD/raw/nifty.csv.gz',
               AUTO_DETECT=TRUE, SAMPLE_SIZE=-1)
  WHERE trade_ts >= TIMESTAMP '2023-01-01'
    AND trade_ts <  TIMESTAMP '2023-02-01'
)
```

```
TO '/Volumes/ExtHDD/nifty_parquet'
(WINDOW=TRUE, FORMAT PARQUET, COMPRESSION ZSTD,
 PARTITION_BY (year(trade_ts), month(trade_ts)));
```

If your data is already split per day/table, you can pass a **glob**:

```sql
COPY (
  SELECT *
  FROM read_csv('/Volumes/ExtHDD/raw/2023-01/*.csv.gz', AUTO_DETECT=TRUE)
)
TO '/Volumes/ExtHDD/nifty_parquet'
(FORMAT PARQUET, COMPRESSION ZSTD,
 PARTITION_BY (year(trade_ts), month(trade_ts)));
```

Python (pyarrow/polars) converter for one month

```python
import polars as pl
from pathlib import Path

in_glob = "/Volumes/ExtHDD/raw/2023-01/*.csv"   # or .csv.gz
out_dir = Path("/Volumes/ExtHDD/nifty_parquet")

(
    pl.scan_csv(in_glob, ignore_errors=True, try_parse_dates=True)
      .with_columns([
          pl.col("trade_ts").str.strptime(pl.Datetime, strict=False),
      ])
      .sink_parquet(out_dir, compression="zstd", statistics=True,
                    hive_partitioning=True, partition_by=["year","month"])
)
```

Running the backtest efficiently

- **Fastest IO:** copy the needed month(s) from HDD → internal SSD (e.g., `/tmp/nifty_m06/`) before the run.
- **Direct read is okay** if you accept HDD speed; use Polars' lazy scan to read only needed columns/rows.

Load month-by-month into NumPy arrays

```python
import polars as pl
import numpy as np
from glob import glob

month_paths = glob("/Volumes/ExtHDD/nifty_parquet/year=2023/month=01/*.parquet")

# Read only needed columns; predicate pushdown reduces IO
df = pl.scan_parquet(month_paths).select([
    "trade_ts", "symbol", "strike", "ltp", "oi", "volume"
]).collect()

# Convert to NumPy for your TA-Lib pipeline
ts = df["trade_ts"].to_numpy()
ltp = df["ltp"].to_numpy(dtype=np.float32)
oi  = df["oi"].to_numpy(dtype=np.int32)
# ... run TA-Lib / Numba JIT on these arrays
```

For **6-month runs**, just build a list of 6 partitions and concat in Polars (still columnar, still pushdown), then materialize to NumPy.

What to expect performance-wise (HDD vs internal SSD)

- **HDD** (120–180 MB/s): Converting one month of raw → Parquet is mostly **CPU-bound** on parsing + compression; HDD won't kill you, just don't thrash random IO. Expect a few–several minutes per month depending on month size.
- **Running backtests**: If you copy 6 months (say **~30–60 GB Parquet**) to internal SSD first, your loop will be much snappier. If you run directly from HDD, it's still fine for **sequential scans** but slower for repeated passes.

Concrete plan for you (no SSD purchase)

1. Use the **external HDD** as bulk storage:
   - Restore/extract there.
   - Convert **month-by-month** to Parquet (ZSTD), partitioned by `year=/month=`.
   - Delete raw files for months you've converted to keep space down.
2. When you want to backtest:
   - **Copy** the target month(s) Parquet folders to your internal disk (stay ⩽100 GB).

- Run your NumPy/TA-Lib pipeline on that data.
- Delete the local copy after you're done.

If you share:

- rows/columns schema (names + types),
- typical month size (rows or GB),
- whether your raw is CSV or MariaDB files,

…I'll give you a drop-in **DuckDB or Polars script** tailored to your schema that converts and partitions with minimal temp space and a sample **month-by-month backtest driver** that keeps you under 100 GB.

> Ok, suggest me this for this plan like I have all these 10 parts of the files. They are all like 1010 GB and the last file is like 2.3 GB. Now let's say I want to take out the data from the 2.3 GB one and then run backtest on that one for now just for the sake of structure of data and other stuff.
>
> So how would I go about that? Right now the files are on the hard drive. So should I process it there? Should I copy that to Mac or how to about it?

Since you just want to use that **last 2.3 GB part** to figure out the **data structure** and **set up your backtest code**, here's the cleanest and safest way:

Step 1 — Keep It on the External Drive for Now

- No need to waste internal space yet.
- You can extract/process *directly* from the external HDD just to explore the schema and logic.
- Performance won't matter much for this one file.

Step 2 — Extract Just That Last Part

A `.7z.010` **isn't a standalone archive** — it's the *end* of the multi-part set.
If the table/data you want happens to be in that part, you **still need the earlier parts**

present in the same folder for extraction, because 7-Zip needs the full chain to unpack.

If you're not sure which file/table is in which part:

- You can *list* archive contents without extracting:

```bash
7z l das_niftyopt_backup.7z.001
```

You'll see the filenames inside and their sizes.
If you spot something small enough (e.g., a particular month/day table), you can extract only that file:

```bash
7z x das_niftyopt_backup.7z.001 desired_table.sql -o/Volumes/YourDrive/temp
```

> `-o` specifies output folder — you can point it to a folder on your Mac's internal disk if you want faster DB/CSV parsing.

Step 3 — Process on the Mac (Better for Speed)

- If the extracted file is **under ~5–10 GB**, copy it to your Mac's internal SSD.
- Run your parsing → backtest code locally — much faster than reading from HDD.
- This is especially true if you're converting to Parquet, loading into NumPy, or running TA-Lib — all benefit from SSD speed.

Step 4 — Inspect & Build Your Pipeline

From that small chunk you can:

1. See if it's a `.sql` dump → `mysqldump` style table structure + inserts.
   - Import to a **temporary local MariaDB** (can run from Docker or Homebrew).
   - Export to CSV or pipe directly into Python.
2. See if it's already CSV — just load into Polars/Pandas.
3. Build your **month-by-month conversion & backtest script** based on this.

Example CSV → Parquet on Mac:

```python
import polars as pl

df = pl.read_csv("/path/to/smallfile.csv")
df.write_parquet("/path/to/smallfile.parquet", compression="zstd")
```

Step 5 — For Later Big Runs

Once the code is working:

- **Copy** the target month's raw file(s) from HDD → Mac SSD before processing.
- Keep Parquet master copy on HDD; only bring months you need to internal.
- Process in **batches** to stay under your 100 GB cap.

💡 **For your immediate test:**

1. List archive contents (`7z l`) from HDD.
2. Extract just a small file/table to Mac.
3. Parse it locally and get your backtest pipeline working.

If you want, I can give you a **one-shot command** that:

- Reads from the multi-part archive on HDD
- Extracts only the file matching your desired month/day
- Saves it to your Mac's `/tmp/` for testing

Do you want me to prepare that exact command for you?

Yes, I would like to fetch all the data from July 2025 available in that last file. And I would like it not in the temporary folder but the folder where I'm writing this for now. I can delete it later.

Alright — here's exactly how I'd pull **only July 2025 data** from your multi-part `.7z` archive while keeping it in your **current working folder**.

1️⃣ Check what's inside

First, list the archive contents so we know how July 2025 is stored (folder name, table name, file naming convention, etc.):

```bash
cd /path/to/your/archive/folder   # where all das_niftyopt_backup.7z.001–.010 are
7z l das_niftyopt_backup.7z.001 | less
```

- Look for any files whose name contains **2025-07**, **202507**, or similar.
- Note the exact relative path inside the archive.

Example output might show something like:

```bash
2025-07/niftyopt_2025-07-01.sql
2025-07/niftyopt_2025-07-02.sql
...
```

2️⃣ Extract only the July 2025 files

Once you know the pattern, you can extract *only* those files directly to your **current folder**:

```bash
7z x das_niftyopt_backup.7z.001 '*2025-07*'
```

or, if the pattern is numeric:

```bash
7z x das_niftyopt_backup.7z.001 '*202507*'
```

Notes:

- The `*` wildcards must be quoted (`'`) so your shell doesn't expand them.
- This works even if the data is in `.7z.010`, because `7z` will automatically chain through all `.001` – `.010` parts.
- Files will appear in your current working directory (preserving any subfolder paths from inside the archive).

3️⃣ After extraction

- You'll have only July 2025's SQL/CSV files here.
- You can now:
  - Import into MariaDB locally.
  - Or parse directly in Python / convert to Parquet for your backtest.

4️⃣ Example: convert to Parquet for quick test

```python
import polars as pl
import glob

files = glob.glob("./*2025-07*.csv")  # or .sql converted to CSV
df = pl.read_csv(files)
df.write_parquet("july2025.parquet", compression="zstd")
```

⚠️ If the July 2025 data is **inside a single huge table file** (not split by month), then we can't filter by filename — we'd need to **pipe-extract and filter rows on the fly** without fully inflating it.
That's a slightly different command but still doable.

Do you want me to prepare the **row-level streaming filter** too, just in case July 2025 is buried inside a bigger SQL dump rather than split into separate files? That way you won't have to uncompress the entire 600 GB just to isolate one month.