```
# pip install databento

import databento as db
import pandas as pd
import numpy as np
import time
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo
```

```
# API_KEY = "db-iKVuPA7sBdWpefhWQyHPkrSjJpLgH"
# client = db.Historical(API_KEY)
# tickers = ["ES.FUT"]


# def download_day(tracking_day):
#     print(tracking_day)
#     chi = ZoneInfo("America/Chicago")
#     day = datetime.fromisoformat(tracking_day).replace(tzinfo=chi)
#     start_dt = (day - timedelta(days=1)).replace(hour=17, minute=00, second=00, mi-
crosecond=0)
#     end_dt   = day.replace(hour=16, minute=00, second=00, microsecond=0)

#     start_time = start_dt.isoformat()
#     end_time   = end_dt.isoformat()

#     # download raw data
#     for symbol in tickers:
#         print(f"Downloading {symbol}...")

#         quotes = client.timeseries.get_range(
#         dataset="GLBX.MDP3",
#         schema="tbbo",
#         symbols=tickers,
#         stype_in="parent",
#         start=start_time,
#         end=end_time,
#         ).to_df()

#         trades = client.timeseries.get_range(
#         dataset="GLBX.MDP3",
#         schema="trades",
#         symbols=tickers,
#         stype_in="parent",
#         start=start_time,
#         end=end_time,
#         ).to_df()
```

```
#     ### we keep only the futures contracts (get rid of spreads)
#     mask1 = quotes["symbol"].str.match(r"^[A-Z]{2}[HMUZ][0-9]$")
#     mask2 = trades["symbol"].str.match(r"^[A-Z]{2}[HMUZ][0-9]$")

#     quotes = quotes[mask1]
#     trades = trades[mask2]

#     ### we get the bid and ask value every 15 min
#     quotes = quotes.rename(columns={
#         'ts_event': 'Time'
#     })
#     quotes['Time'] = pd.to_datetime(quotes['Time'], errors='coerce')
#     quotes = quotes.dropna(subset=['Time'])
#     quotes['Time'] = quotes['Time'].dt.floor('15min')
#     quotes.set_index('Time', inplace=True)
#     quotes = quotes[['bid_px_00', 'ask_px_00', 'symbol']]
#     quotes = quotes.groupby([quotes.index, 'symbol']).agg({
#         'bid_px_00' : 'first',
#         'ask_px_00' : 'first',
#     })


#     ### we get the vwap since opening every 15 min
#     trades = trades.rename(columns={
#         'ts_event': 'Time'
#     })

#     trades['Time'] = pd.to_datetime(trades['Time'], errors='coerce')
#     trades = trades.dropna(subset=['Time'])
#     trades['Time'] = trades['Time'].dt.floor('15min')
#     trades.set_index('Time', inplace=True)

#     trades['cum_notional'] = (trades['price'] * trades['size']).cumsum()
#     trades['cum_volume']   = trades['size'].cumsum()
#     trades['vwap_cum']     = trades['cum_notional'] / trades['cum_volume']

#     trades = trades[['vwap_cum', 'symbol']]
#     trades = trades.groupby([trades.index, 'symbol']).agg({
#         'vwap_cum' : 'first'
#     })

#     ### we merge the two tables together
#     trades.reset_index(inplace=True)
#     quotes.reset_index(inplace=True)
#     df = pd.merge(trades,
```

```
#                         quotes,
#                         on = ['Time', 'symbol']
#                         )



#     ### we compute the time until expiry to rank the contracts
#     mmap = {'F':1,'G':2,'H':3,'J':4,'K':5,'M':6,'N':7,'Q':8,'U':9,'V':10,'X':11,'Z'
:12}
#     m = df['symbol'].str.extract(r'([FGHJKMNQUVXZ])(\d{1,2})$')
#     df['exp_month'] = m[0].map(mmap).astype('Int64')
#     y = m[1].astype(int)
#     df['exp_year']  = np.where(y < 10, 2020 + y, 2000 + y)   # 4→2024, 24→2024

#     base = pd.to_datetime(df['exp_year'].astype(str) + df['exp_-
month'].astype(str).str.zfill(2), format='%Y%m')
#     third_fri = base + pd.offsets.WeekOfMonth(week=2, week-
day=4)  # 0=1st Fri, 2=3rd Fri
#     df['expiry_dt_et'] = third_fri.dt.tz_localize('America/New_York') + pd.-
Timedelta(hours=9, minutes=30)

#     df['time_to_expiry'] = df['expiry_dt_et'] - df['Time']
#     df['days_to_expiry'] = df['time_to_expiry'].dt.total_seconds() / 86400

#     df['expiry_order'] = df.groupby('Time')
['expiry_dt_et'].rank(method='dense').astype(int)
#     df = df[['Time', 'symbol', 'expiry_order', 'bid_px_00', 'ask_px_00', 'vwap_cum'
]]



#     ### we create empty row for every missing contracts at some point in time
#     times   = pd.to_datetime(df['Time'])
#     symbols = df['symbol'].unique()
#     full_ix = pd.MultiIndex.from_product([times.unique(), symbols], names=
['Time','symbol'])

#     g = (df.assign(Time=times)
#         .set_index(['Time','symbol'])
#         .reindex(full_ix)
#         .sort_index())

#     ### we fill the value of the bid and ask forward, same for the vwap
#     df_filled = (g.groupby(level='symbol').ffill()
#                 .reset_index())

#     df_filled.to_csv(f'/data/workspace_files/data/{tracking_day}.csv')
```

```
# trading_day = "2024-05-10"
# download_day(trading_day)
```

```
2024-05-10
Downloading ES.FUT...
/tmp/ipykernel_646/3795707557.py:97: PerformanceWarning: Non-vectorized DateOffset bei
  third_fri = base + pd.offsets.WeekOfMonth(week=2, weekday=4)  # 0=1st Fri, 2=3rd Fri
```

```
# from time import sleep

# def run_range(download_fn, start="2010-06-06", end="2010-06-10", business_-
days=True, pause_sec=0.0):
#     # Choose trading days: Mon-Fri (business) or every calendar day
#     rng = (pd.bdate_range if business_days else pd.date_range)
(start=start, end=end, freq="D")
#     failures = []

#     total = len(rng)
#     for i, d in enumerate(rng, 1):
#         td = d.date().isoformat()
#         try:
#             print(f"[{i}/{total}] {td} ... ", end="", flush=True)
#             download_fn(td)
#             print("done")
#         except Exception as e:
#             print("FAILED")
#             failures.append((td, str(e)))
#         if pause_sec:
#             sleep(pause_sec)

#     print(f"\nFinished: {total - len(failures)} ok, {len(failures)} failed.")
#     if failures:
#         print("Sample failures (up to 10):")
#         for td, msg in failures[:10]:
#             print(f"  {td}: {msg}")

# # Run it
# run_range(download_day, start="2010-06-06", end="2025-01-01", business_-
days=True, pause_sec=0.0)
```

```
from __future__ import annotations

import re
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo

import numpy as np
```

```
import pandas as pd
import databento as db


# ===== CONFIG =====
API_KEY = "db-iKVuPA7sBdWpefhWQyHPkrSjJpLgH"
client = db.Historical(API_KEY)

# Parent symbol -> expands to all ES contracts
PARENT = "ES.FUT"

# Month code mapping
MONTH_MAP = {'F':1,'G':2,'H':3,'J':4,'K':5,'M':6,'N':7,'Q':8,'U':9,'V':10,'X':11,'Z':
12}
SYM_RE = re.compile(r"^([A-Z]{1,3})([FGHJKMNQUVXZ])(\d{1,2})$")  # e.g. ESZ4, ESH25


def _resolve_year(two_digit: int, ref_year: int) -> int:
    """
    Map a 1-2 digit year code to a full year close to the reference year (track-
ing day).
    Works across 2010s/2020s and removes the earlier hack that caused wrong years.
    """
    candidates = [2000 + two_digit, 2010 + two_digit, 2020 + two_digit, 2030 + two_di
git]
    return min(candidates, key=lambda Y: (abs(Y - ref_year), Y))


def _parse_expiry(symbol: str, ref_year: int) -> tuple[int, int]:
    """
    Return (YYYY, MM) for a futures symbol like ESZ4 / ESZ24.
    """
    m = SYM_RE.match(symbol)
    if not m:
        return (9999, 12)  # push unknowns to the back
    _, mon_code, yy = m.groups()
    mm = MONTH_MAP[mon_code]
    y = int(yy)
    yyyy = _resolve_year(y, ref_year)
    return (yyyy, mm)


def _build_bar_ends(start_dt, end_dt, freq="15min"):
    """
    Build 15-min bar end timestamps from session open to session close inclusive,
    in America/Chicago timezone.
    First bar ends at start_dt + 15min.
```

```
    """
    start_dt = pd.Timestamp(start_dt)
    end_dt   = pd.Timestamp(end_dt)
    first = (start_dt + pd.Timedelta(minutes=15)).tz_convert(start_dt.tz)
    return pd.date_range(first, end_dt, freq=freq)


def download_day(tracking_day: str, output_csv: str | None = None) -> pd.DataFrame:
    """
    Download TBBO + trades for the ES futures session that runs from
    17:00 CT (prev day) to 16:00 CT (tracking_day), compute:
      - Bid/Ask at each 15-min bar end (last known TBBO)
      - VWAP since session open at each 15-min bar end
    Keep only ONE contract at each timestamp: the front month (nearest expiry).
    Save to CSV if output_csv is provided. Return the final DataFrame.
    """
    chi = ZoneInfo("America/Chicago")
    day = datetime.fromisoformat(tracking_day).replace(tzinfo=chi)
    start_dt = (day - timedelta(days=1)).replace(hour=17, minute=0, second=0, mi-
crosecond=0)
    end_dt   = day.replace(hour=16, minute=0, second=0, microsecond=0)

    # Build 15-minute bar end grid in Chicago time
    bar_ends = _build_bar_ends(start_dt, end_dt, freq="15min")

    # === Download raw data (expand parent to all active ES contracts) ===
    quotes = client.timeseries.get_range(
        dataset="GLBX.MDP3",
        schema="tbbo",
        symbols=[PARENT],
        stype_in="parent",
        start=start_dt.isoformat(),
        end=end_dt.isoformat(),
    ).to_df()

    trades = client.timeseries.get_range(
        dataset="GLBX.MDP3",
        schema="trades",
        symbols=[PARENT],
        stype_in="parent",
        start=start_dt.isoformat(),
        end=end_dt.isoformat(),
    ).to_df()

    if quotes.empty and trades.empty:
        raise RuntimeError("No data returned for the requested session.")
```

```
    # === Keep only outright futures, drop inter-month spreads ===
    # ESZ4, ESH25, etc. (2-3 letters root + month code + 1-2 digits year)
    mask_q = quotes["symbol"].astype(str).str.match(SYM_RE)
    mask_t = trades["symbol"].astype(str).str.match(SYM_RE)
    quotes = quotes[mask_q].copy()
    trades = trades[mask_t].copy()

    # === Normalize timestamps to America/Chicago ===
    # Databento returns tz-aware UTC; convert to CT for correct session math
    quotes["Time"] = pd.to_datetime(quotes["ts_event"], utc=True).dt.tz_convert(chi)
    trades["Time"] = pd.to_datetime(trades["ts_event"], utc=True).dt.tz_convert(chi)

    # === Prepare TBBO at 15-min bar ends: last known be-
fore or at each bar end (per symbol) ===
    def resample_tbbo_per_symbol(df_sym: pd.DataFrame) -> pd.DataFrame:
        q = (
            df_sym.loc[:, ["Time", "bid_px_00", "ask_px_00"]]
            .sort_values("Time")
            .drop_duplicates("Time", keep="last")
        )
        # asof needs both sides sorted and unique on the merge key
        target = pd.DataFrame({"Time": bar_ends})
        out = pd.merge_asof(
            target, q, on="Time", direction="backward", allow_exact_matches=True
        )
        out["symbol"] = df_sym["symbol"].iloc[0]
        return out

    tbbo = (
        quotes.sort_values(["symbol", "Time"])
        .groupby("symbol", group_keys=False)
        .apply(resample_tbbo_per_symbol)
    )

    # === VWAP since session open at each bar end: compute cumula-
tives first, then asof ===
    if trades.empty:
        # No trades: vwap is NaN; keep quotes only
        vwap_bars = tbbo.loc[:, ["Time", "symbol"]].copy()
        vwap_bars["vwap_cum"] = np.nan
    else:
        tr = trades.loc[:, ["symbol", "Time", "price", "size"]].copy()
        tr = tr.sort_values(["symbol", "Time"])
        tr["notional"] = tr["price"] * tr["size"]
        # cumulative within the session, per symbol
        tr["cum_notional"] = tr.groupby("symbol")["notional"].cumsum()
        tr["cum_volume"]   = tr.groupby("symbol")["size"].cumsum()
```

```
        tr = tr.loc[:, ["symbol", "Time", "cum_notional", "cum_volume"]]

        def vwap_per_symbol(df_sym: pd.DataFrame) -> pd.DataFrame:
            s = df_sym.loc[:, ["Time", "cum_notional", "cum_volume"]].drop_dupli-
cates("Time")
            s = s.sort_values("Time")
            target = pd.DataFrame({"Time": bar_ends})
            out = pd.merge_asof(
                target, s, on="Time", direction="backward", allow_exact_matches=True
            )
            out["symbol"] = df_sym["symbol"].iloc[0]
            out["vwap_cum"] = out["cum_notional"] / out["cum_volume"]
            return out.loc[:, ["Time", "symbol", "vwap_cum"]]

        vwap_bars = (
            tr.groupby("symbol", group_keys=False)
              .apply(vwap_per_symbol)
        )

    # === Merge TBBO and VWAP on (Time, symbol) ===
    bars = pd.merge(
        tbbo, vwap_bars, on=["Time", "symbol"], how="left"
    )

    # === Select ONLY ONE product: the front month at each timestamp ===
    ref_year = day.year
    exp_ym = bars["symbol"].astype(str).map(lambda s: _parse_expiry(s, ref_year))
    bars["exp_year"]  = [ym[0] for ym in exp_ym]
    bars["exp_month"] = [ym[1] for ym in exp_ym]
    bars["expiry_key"] = bars["exp_year"] * 12 + bars["exp_month"]


    out = bars.sort_values(["Time", "expiry_key"]).reset_index(drop=True)
    out = out.loc[:, ["Time", "symbol", "bid_px_00", "ask_px_00", "vwap_cum"]]
    # If there were no trades up to a bar, vwap is NaN; forward-fill with-
in the day is reasonable
    # but we only fill *within the same symbol* (already one symbol per row), so:
    out["vwap_cum"] = out["vwap_cum"].ffill()

    # Optional: drop a trailing bar if it's exactly at the close and every-
thing is unchanged
    # (comment this out if you prefer to keep the 16:00 stamp)
    # if len(out) and out["Time"].iloc[-1] == pd.Timestamp(end_dt).tz_convert(chi):
    #     out = out.iloc[:-1].copy()

    if output_csv:
        out.to_csv(output_csv, index=False)
```

```
    return out


# ===== Example usage =====
# df = download_day("2010-06-07", output_csv="/data/workspace_files/new_data/2010-06-
07.csv")
# display(df.head())
```

```
/tmp/ipykernel_646/832639544.py:128: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(resample_tbbo_per_symbol)
/tmp/ipykernel_646/832639544.py:158: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(vwap_per_symbol)
```

```
from time import sleep
import pandas as pd
from pathlib import Path


def run_range(download_fn, start="2010-06-06", end="2010-06-10", business_-
days=True, pause_sec=0.0):

    # Choose trading days: Mon-Fri (business) or every calendar day
    rng = (pd.bdate_range if business_days else pd.date_range)
(start=start, end=end, freq="D")
    failures = []

    total = len(rng)
    for i, d in enumerate(rng, 1):
        td = d.date().isoformat()
        try:
            print(f"[{i}/{total}] {td} ... ", end="", flush=True)
            download_fn(td)   # <- call the download_day function
            print("done")
        except Exception as e:
            print("FAILED")
            failures.append((td, str(e)))
        if pause_sec:
            sleep(pause_sec)

    print(f"\nFinished: {total - len(failures)} ok, {len(failures)} failed.")
    if failures:
        print("Sample failures (up to 10):")
        for td, msg in failures[:10]:
            print(f"  {td}: {msg}")


# === Example run ===
```

```python
# This will loop from 2010-06-06 to 2025-01-01 and call your download_day()
# one date at a time, weekdays only.


outdir = Path("/data/workspace_files/new_data")
outdir.mkdir(parents=True, exist_ok=True)

run_range(
    lambda d: download_day(d, output_csv=str(outdir / f"{d}.csv")),
    start="2010-06-06",
    end="2025-01-01",
    business_days=True,
    pause_sec=0.0,
)
```

```
[1/5324] 2010-06-06 ... FAILED
[2/5324] 2010-06-07 ... done
[3/5324] 2010-06-08 ... done
[4/5324] 2010-06-09 ... done
[5/5324] 2010-06-10 ... done
[6/5324] 2010-06-11 ... done
[7/5324] 2010-06-12 ... FAILED
[8/5324] 2010-06-13 ... FAILED
[9/5324] 2010-06-14 ... done
[10/5324] 2010-06-15 ... done
[11/5324] 2010-06-16 ... done
[12/5324] 2010-06-17 ... done
[13/5324] 2010-06-18 ... done
[14/5324] 2010-06-19 ... FAILED
[15/5324] 2010-06-20 ... FAILED
[16/5324] 2010-06-21 ... done
[17/5324] 2010-06-22 ... done
[18/5324] 2010-06-23 ... done
[19/5324] 2010-06-24 ... done
[20/5324] 2010-06-25 ... done
```

```
  .apply(vwap_per_symbol)
/tmp/ipykernel_646/832639544.py:77: BentoWarning: No data found for the request you su
  quotes = client.timeseries.get_range(
/tmp/ipykernel_646/832639544.py:86: BentoWarning: No data found for the request you su
  trades = client.timeseries.get_range(
/tmp/ipykernel_646/832639544.py:77: BentoWarning: No data found for the request you su
  quotes = client.timeseries.get_range(
/tmp/ipykernel_646/832639544.py:86: BentoWarning: No data found for the request you su
  trades = client.timeseries.get_range(
/tmp/ipykernel_646/832639544.py:128: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(resample_tbbo_per_symbol)
/tmp/ipykernel_646/832639544.py:158: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(vwap_per_symbol)
/tmp/ipykernel_646/832639544.py:128: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(resample_tbbo_per_symbol)
/tmp/ipykernel_646/832639544.py:158: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(vwap_per_symbol)
/tmp/ipykernel_646/832639544.py:128: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(resample_tbbo_per_symbol)
/tmp/ipykernel_646/832639544.py:158: DeprecationWarning: DataFrameGroupBy.apply operat
```

```python
from __future__ import annotations

import re
from datetime import datetime, timedelta, time as dtime
from zoneinfo import ZoneInfo

import numpy as np
import pandas as pd
import databento as db


# ===== CONFIG =====
API_KEY = "db-iKVuPA7sBdWpefhWQyHPkrSjJpLgH"
client = db.Historical(API_KEY)

# Parent symbol -> expands to all ES contracts
PARENT = "ES.FUT"

# Month code mapping & symbol regex (e.g., ESZ4, ESH25)
MONTH_MAP = {'F':1,'G':2,'H':3,'J':4,'K':5,'M':6,'N':7,'Q':8,'U':9,'V':10,'X':11,'Z':
12}
SYM_RE = re.compile(r"^([A-Z]{1,3})([FGHJKMNQUVXZ])(\d{1,2})$")  # root + month + 1-
2 digit year


def _resolve_year(two_digit: int, ref_year: int) -> int:
```

```
    """
    Map a 1-2 digit year code to a full year close to the reference year (track-
ing day).
    """
    candidates = [2000 + two_digit, 2010 + two_digit, 2020 + two_digit, 2030 + two_di
git]
    return min(candidates, key=lambda Y: (abs(Y - ref_year), Y))


def _parse_expiry(symbol: str, ref_year: int) -> tuple[int, int]:
    """
    Return (YYYY, MM) for a futures symbol like ESZ4 / ESZ24.
    """
    m = SYM_RE.match(symbol)
    if not m:
        return (9999, 12)  # push unknowns to the back
    _, mon_code, yy = m.groups()
    mm = MONTH_MAP[mon_code]
    y = int(yy)
    yyyy = _resolve_year(y, ref_year)
    return (yyyy, mm)


def _build_bar_ends(start_dt, end_dt, freq="15min"):
    """
    Build bar-end timestamps from RTH open to RTH close inclusive, in America/Chica-
go time.
    Bars are shifted by +3 minutes relative to the regular grid.
    First bar ends at start_dt + 3 minutes (e.g. 08:33 CT), then every `freq` there-
after.
    Last bar is clipped so it does not exceed end_dt (15:00 CT).
    """
    start_dt = pd.Timestamp(start_dt)
    end_dt   = pd.Timestamp(end_dt)

    # Build grid: start at RTH open + 3min, then every `freq`
    first = start_dt + pd.Timedelta("3min")
    grid = pd.date_range(first, end_dt + pd.Timedelta(freq), freq=freq)

    # Clip any value that overshoots the session close (e.g., 15:03 -> 15:00)
    grid = grid.where(grid <= end_dt, end_dt)

    # Drop duplicates if last two collapse at 15:00
    grid = grid.drop_duplicates()

    return grid
```

```
def download_us_rth_day(tracking_day: str,
                        freq: str = "15min",
                        output_csv: str | None = None) -> pd.DataFrame:
    """
    Download TBBO + trades for ES during **US Regular Trading Hours** (09:30-
16:00 ET),
    compute:
      - Bid/Ask at each bar end (last known TBBO)
      - VWAP since RTH open (cumulative from 09:30 ET) at each bar end
    Keep only ONE contract at each timestamp: the front month (nearest expiry).
    Save to CSV if output_csv is provided. Return the final DataFrame.
    """
    chi = ZoneInfo("America/Chicago")
    nyc = ZoneInfo("America/New_York")

    # Tracking day in CT (we anchor on the local date)
    day_ct = datetime.fromisoformat(tracking_day).replace(tzinfo=chi)

    # US RTH window: 09:30-16:00 ET == 08:30-15:00 CT for the same civil day
    rth_open_ct  = day_ct.replace(hour=8, minute=30, second=0, microsecond=0)
    rth_close_ct = day_ct.replace(hour=15, minute=0, second=0, microsecond=0)

    # Bar ends grid in CT, within RTH only
    bar_ends = _build_bar_ends(rth_open_ct, rth_close_ct, freq=freq)

    # === Download raw data (expand parent to all active ES contracts) with-
in RTH window ===
    quotes = client.timeseries.get_range(
        dataset="GLBX.MDP3",
        schema="tbbo",
        symbols=[PARENT],
        stype_in="parent",
        start=rth_open_ct.isoformat(),
        end=rth_close_ct.isoformat(),
    ).to_df()

    trades = client.timeseries.get_range(
        dataset="GLBX.MDP3",
        schema="trades",
        symbols=[PARENT],
        stype_in="parent",
        start=rth_open_ct.isoformat(),
        end=rth_close_ct.isoformat(),
    ).to_df()
```

```
    if quotes.empty and trades.empty:
        raise RuntimeError("No data returned for the requested RTH window.")

    # === Keep only outright futures, drop inter-month spreads ===
    mask_q = quotes["symbol"].astype(str).str.match(SYM_RE)
    quotes = quotes[mask_q].copy()
    if not trades.empty:
        mask_t = trades["symbol"].astype(str).str.match(SYM_RE)
        trades = trades[mask_t].copy()

    # === Normalize timestamps to America/Chicago ===
    quotes["Time"] = pd.to_datetime(quotes["ts_event"], utc=True).dt.tz_convert(chi)
    if not trades.empty:
        trades["Time"] = pd.to_datetime(trades["ts_event"], utc=True).dt.tz_con-
vert(chi)

    # === Prepare TBBO at bar ends: last known before or at each bar end (per sym-
bol) ===
    def resample_tbbo_per_symbol(df_sym: pd.DataFrame) -> pd.DataFrame:
        q = (
            df_sym.loc[:, ["Time", "bid_px_00", "ask_px_00"]]
            .sort_values("Time")
            .drop_duplicates("Time", keep="last")
        )
        target = pd.DataFrame({"Time": bar_ends})
        out = pd.merge_asof(
            target, q, on="Time", direction="backward", allow_exact_matches=True
        )
        out["symbol"] = df_sym["symbol"].iloc[0]
        return out

    tbbo = (
        quotes.sort_values(["symbol", "Time"])
              .groupby("symbol", group_keys=False)
              .apply(resample_tbbo_per_symbol)
    )

    # === VWAP since RTH open: cumulative only within the RTH window (per symbol) ===
    if trades.empty:
        vwap_bars = tbbo.loc[:, ["Time", "symbol"]].copy()
        vwap_bars["vwap_cum"] = np.nan
    else:
        tr = trades.loc[:, ["symbol", "Time", "price", "size"]].copy()
        tr = tr.sort_values(["symbol", "Time"])
        tr["notional"] = tr["price"] * tr["size"]
        # start cumulation at RTH open by filtering to the RTH window only
        tr = tr[(tr["Time"] >= rth_open_ct) & (tr["Time"] <= rth_close_ct)]
```

```
        tr["cum_notional"] = tr.groupby("symbol")["notional"].cumsum()
        tr["cum_volume"]   = tr.groupby("symbol")["size"].cumsum()
        tr = tr.loc[:, ["symbol", "Time", "cum_notional", "cum_volume"]]

        def vwap_per_symbol(df_sym: pd.DataFrame) -> pd.DataFrame:
            s = df_sym.loc[:, ["Time", "cum_notional", "cum_volume"]].drop_dupli-
cates("Time")
            s = s.sort_values("Time")
            target = pd.DataFrame({"Time": bar_ends})
            out = pd.merge_asof(
                target, s, on="Time", direction="backward", allow_exact_matches=True
            )
            out["symbol"] = df_sym["symbol"].iloc[0]
            out["vwap_cum"] = out["cum_notional"] / out["cum_volume"]
            return out.loc[:, ["Time", "symbol", "vwap_cum"]]

        vwap_bars = tr.groupby("symbol", group_keys=False).apply(vwap_per_symbol)

    # === Merge TBBO and VWAP on (Time, symbol) ===
    bars = pd.merge(tbbo, vwap_bars, on=["Time", "symbol"], how="left")

    # === Determine front month (nearest expiry) per timestamp and keep only that ===
    ref_year = day_ct.year
    exp_ym = bars["symbol"].astype(str).map(lambda s: _parse_expiry(s, ref_year))
    bars["exp_year"]  = [ym[0] for ym in exp_ym]
    bars["exp_month"] = [ym[1] for ym in exp_ym]
    bars["expiry_key"] = bars["exp_year"] * 12 + bars["exp_month"]



    # IMPORTANT: columns indexer must be a LIST (not a tuple)
    out = bars.loc[:, ["Time", "symbol", "bid_px_00", "ask_px_00", "vwap_cum", "ex-
p_year", "exp_month", "expiry_key"]].sort_values("Time")



    # Forward-fill VWAP after first valid trade within the day
    # (initial NaNs—before first trade—are left as NaN)
    first_valid = out["vwap_cum"].first_valid_index()
    if first_valid is not None:
        out.loc[first_valid:, "vwap_cum"] = out.loc[first_valid:, "vwap_cum"].ffill()

    if output_csv:
        out.to_csv(output_csv, index=False)

    return out
```

```
from time import sleep
import pandas as pd
from pathlib import Path

def run_range(download_fn, start="2010-06-06", end="2010-06-10",
              business_days=True, pause_sec=0.0):
    # Choose trading days: Mon-Fri (business) or every calendar day
    rng = (pd.bdate_range if business_days else pd.date_range)(
        start=start, end=end, freq="D"
    )
    failures = []
    total = len(rng)

    for i, d in enumerate(rng, 1):
        td = d.date().isoformat()
        try:
            print(f"[{i}/{total}] {td} ... ", end="", flush=True)
            download_fn(td)
            print("done")
        except Exception as e:
            print("FAILED")
            failures.append((td, str(e)))
        if pause_sec:
            sleep(pause_sec)

    print(f"\nFinished: {total - len(failures)} ok, {len(failures)} failed.")
    if failures:
        print("Sample failures (up to 10):")
        for td, msg in failures[:10]:
            print(f"  {td}: {msg}")

# === Example run ===
outdir = Path("/data/workspace_files/us_data")
outdir.mkdir(parents=True, exist_ok=True)

# If your function is named download_us_rth_day (per the latest code), use that:
run_range(
    lambda d: download_us_rth_day(d, freq="15min",
                                  output_csv=str(outdir / f"{d}.csv")),
    start="2010-06-07",
    end="2025-01-01",
    business_days=True,
    pause_sec=0.0,
)
```

```
# You should now see files like:
# /data/workspace_files/2010-06-07_RTH.csv
# /data/workspace_files/2010-06-08_RTH.csv
[1/5323] 2010-06-07 ... done
[2/5323] 2010-06-08 ... done
[3/5323] 2010-06-09 ... done
[4/5323] 2010-06-10 ... done
[5/5323] 2010-06-11 ... done
[6/5323] 2010-06-12 ... FAILED
[7/5323] 2010-06-13 ... FAILED
[8/5323] 2010-06-14 ... done
[9/5323] 2010-06-15 ... done
[10/5323] 2010-06-16 ... done
[11/5323] 2010-06-17 ... done
[12/5323] 2010-06-18 ... done
[13/5323] 2010-06-19 ... FAILED
[14/5323] 2010-06-20 ... FAILED
[15/5323] 2010-06-21 ... done
[16/5323] 2010-06-22 ... done
[17/5323] 2010-06-23 ... done
[18/5323] 2010-06-24 ... done
[19/5323] 2010-06-25 ... done
[20/5323] 2010-06-26 ... FAILED
  vwap_bars = tr.groupby("symbol", group_keys=False).apply(vwap_per_symbol)
/tmp/ipykernel_688/812558982.py:145: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(resample_tbbo_per_symbol)
/tmp/ipykernel_688/812558982.py:173: DeprecationWarning: DataFrameGroupBy.apply operat
  vwap_bars = tr.groupby("symbol", group_keys=False).apply(vwap_per_symbol)
/tmp/ipykernel_688/812558982.py:95: BentoWarning: No data found for the request you su
  quotes = client.timeseries.get_range(
/tmp/ipykernel_688/812558982.py:104: BentoWarning: No data found for the request you s
  trades = client.timeseries.get_range(
/tmp/ipykernel_688/812558982.py:95: BentoWarning: No data found for the request you su
  quotes = client.timeseries.get_range(
/tmp/ipykernel_688/812558982.py:104: BentoWarning: No data found for the request you s
  trades = client.timeseries.get_range(
/tmp/ipykernel_688/812558982.py:145: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(resample_tbbo_per_symbol)
/tmp/ipykernel_688/812558982.py:173: DeprecationWarning: DataFrameGroupBy.apply operat
  vwap_bars = tr.groupby("symbol", group_keys=False).apply(vwap_per_symbol)
/tmp/ipykernel_688/812558982.py:145: DeprecationWarning: DataFrameGroupBy.apply operat
  .apply(resample_tbbo_per_symbol)
/tmp/ipykernel_688/812558982.py:173: DeprecationWarning: DataFrameGroupBy.apply operat
```