

		Low
series for tw	572753906	150.63999938964
eral instrum	18310547	151.16999816894
o-date avera	18310547	150.41999816894
	633789062	149.22000122070

wf_samples.sql
File

wf_createtables.sql
File

just read all the stuff first

Thought for 52s >

Got it — I’ve read everything you provided (the assignment prompt screenshot, the dataset preview screenshot, and both SQL files).

What the assignment is asking (from your screenshot)

1.

Get a dataset that has a time series for 2+ separate items (e.g., multiple stocks/crypto), ideally since Jan 1, 2022.
2.

Use window functions (SQL or dplyr) to compute, for each item:

•

Year-to-date (YTD) average

•

6-day moving average

What your dataset looks like (from your table screenshot)

Your structure is appropriate for this task:

- Item id: Ticker (e.g., AAPL)
- Time: Date
- Values: Open/High/Low/Close/Adj Close/Volume (you’ll pick one “value” column to average, typically Close Or Adj Close)

One thing to double-check: the screenshot shows dates starting 2023-02-07 for AAPL. If the requirement is strict about since 2022-01-01, you’ll need the file to actually include 2022 data (or use a different dataset / pull longer history).

What's inside your SQL files

wf_createtables.sql

- Creates a toy table `afc_east(game_date, home_team, away_team, home_points, away_points)`
- Inserts sample rows
- Creates `teams` from distinct home teams

wf_samples.sql

This is basically a **window function demo pack**, showing patterns you'll reuse for the moving averages:

- Regular `AVG()` vs **window AVG**: `AVG(home_points) OVER()`
- **Partitioned** window AVG: `AVG(home_points) OVER(PARTITION BY home_team)`
- **Ranking**: `RANK() OVER(ORDER BY home_points DESC)` and partitioned rank
- **Cumulative "to date"** pattern (key idea for YTD):

```
... OVER(PARTITION BY home_team ORDER BY game_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```
- `NTILE(3)` example

Those "to date" + "partition by item" patterns are exactly what you'll adapt to:

- `PARTITION BY ticker, year`
- `ORDER BY date`
- frame = **unbounded preceding** → **current row** for YTD
- frame = **5 preceding** → **current row** for 6-day moving average

Small heads-up

I also got a system notice that **some older uploaded files have expired**. Right now I can still see and read the two SQL files you uploaded; if you want me to read any other missing/expired file (like your actual `stocks.csv`), you'll need to upload it again.



```
SELECT round(avg(home_points),1) AS home_avg
FROM afc_east
ORDER BY home_avg DESC;

-- subquery in from clause
SELECT *,
(SELECT round(avg(home_points),1) AS home_avg FROM afc_east),
home_points - (SELECT round(avg(home_points),1) FROM afc_east) AS home_differential
FROM afc_east
ORDER BY home_team, game_date;
```

explain this

Thought for 12s >

Here's what that SQL is doing, line by line.

Query 1

sql

```
SELECT round(avg(home_points),1) AS home_avg
FROM afc_east
ORDER BY home_avg DESC;
```

What it returns

 Copy code

- `avg(home_points)` computes the **overall average** of `home_points` across **all rows** in `afc_east`.
- `round(..., 1)` rounds that average to **1 decimal place**.
- Result is **one row, one column**: `home_avg`.

Important note

- `ORDER BY home_avg DESC` is basically pointless here because the query returns **only one row**. Ordering doesn't change anything.

Query 2 (scalar subquery in SELECT list)

sql

```
SELECT *,
  (SELECT round(avg(home_points),1) AS home_avg FROM afc_east),
  home_points - (SELECT round(avg(home_points),1) FROM afc_east) AS home_differen
FROM afc_east
ORDER BY home_team, game_date;
```

 Copy code

What it returns

This returns **every row** from `afc_east` (`SELECT *`), plus two extra computed columns:

1. `home_avg` column

sql

```
(SELECT round(avg(home_points),1) FROM afc_east)
```

 Copy code

- This is a **scalar subquery**: it returns a single value (the overall average).
- Because it's **not correlated** (it doesn't reference the outer row), it's the **same constant** for every row.

2. `home_differential` column

sql

```
home_points - (SELECT round(avg(home_points),1) FROM afc_east)
```

 Copy code

- For each game row, it computes:
 - **that row's** `home_points`
 - minus the **overall (rounded) average home_points**
- So it tells you whether that game's home points were above/below the dataset average.

Why the subquery appears twice

- You're calculating the same average **two separate times**.
- SQL engines *may* optimize this, but conceptually it's duplicated work.

Cleaner way (preferred): use a window function

This produces the same idea without repeating the subquery:

sql

```
SELECT
    *,
    round(avg(home_points) OVER (), 1) AS home_avg,
    home_points - round(avg(home_points) OVER (), 1) AS home_differential
FROM afc_east
ORDER BY home_team, game_date;
```

- `avg(home_points) OVER ()` = overall average as a window value, repeated on each row.

 Copy code

Accuracy note about rounding

Right now you're subtracting the **rounded** average. That slightly changes the differential.

If you want *more accurate math*, subtract using the full average, and round only for display:

sql

```
SELECT
    *,
    round(avg(home_points) OVER (), 1) AS home_avg_display,
    round(home_points - avg(home_points) OVER (), 1) AS home_differential
FROM afc_east
ORDER BY home_team, game_date;
```

If you want, paste the table schema (or the sample rows) and I'll show exactly what the output columns look like for one or two rows.

 Copy code



explain the rest of the code in wf_sample

Thought for 16s >

Sure — here's **the rest of** `wf_samples.sql` , query by query, with the key idea each one is teaching.

3) Overall average via a window function (no subquery)

sql

```
SELECT *,
  ROUND(AVG(home_points) OVER(),1) AS home_avg,
  ROUND(home_points - AVG(home_points) OVER(),1) AS home_differential
FROM afc_east
ORDER BY home_team, game_date;
```

What `OVER()` changes

 Copy code

- `AVG(home_points)` (no `OVER`) collapses the table into **1 row**.
- `AVG(home_points) OVER()` computes the **same overall average**, but **keeps every row** and repeats that average on each row.

Columns added

- `home_avg` : the **overall** mean of `home_points` (same number repeated)
- `home_differential` : each row's `home_points` - `overall_avg`

✓ This is the “window function replacement” for the scalar subquery you asked about earlier.

4) Average *by team* via `PARTITION BY`

sql

```
SELECT *,
  ROUND(AVG(home_points) OVER(PARTITION BY home_team),1) AS home_avg,
  ROUND(home_points - AVG(home_points) OVER(PARTITION BY home_team),1) AS home_di
```

```
FROM afc_east
ORDER BY home_team, game_date;
```

 Copy code

What `PARTITION BY home_team` does

- Splits the table into groups by `home_team`
- Computes the average **within each team**, then repeats that team's average on that team's rows

Interpretation

- `home_avg` is now **"this team's typical home points"**
- `home_differential` is now **"this game vs this team's own average"**

5) Global rank of `home_points`

sql

```
SELECT *,
  RANK() OVER(ORDER BY home_points DESC) AS home_rank
FROM afc_east
ORDER BY home_team, game_date;
```

What it does

 Copy code

- Ranks **all rows** from highest `home_points` to lowest
- `RANK()` means ties share the same rank, and ranks can skip (e.g., 1, 1, 3)

Important detail

- The `ORDER BY home_team, game_date` at the end is **just display sorting**.
- The ranking logic is controlled by `ORDER BY home_points DESC` **inside** the `OVER(...)`.

6) Rank *within each team*

sql

```
SELECT *,
  RANK() OVER(PARTITION BY home_team ORDER BY home_points DESC) AS home_rank
FROM afc_east
ORDER BY home_team, game_date;
```

 Copy code

What changes vs #5

- Now you get a separate ranking table **per team**
- Each team has its own “best home_points game = rank 1”

So:

- Query #5 answers: “What are the biggest home_point games overall?”
- Query #6 answers: “What are the biggest home_point games **for each team?**”

7) Running (cumulative) total by team over time

sql

```
SELECT *,
  SUM(home_points) OVER(
    PARTITION BY home_team
    ORDER BY game_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS cum_home_points
FROM afc_east
ORDER BY home_team, game_date;
```

This is the “to-date” pattern

For each team:

 Copy code

- Sort rows by game_date
- For each row, sum from the **first row of that team** up to the **current row**

Why the frame matters

- ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW explicitly says:
“running total up to today (this row)”

This is the exact pattern you'll reuse for **YTD** calculations (just switch `SUM` to `AVG`, and partition by year too).

8) `NTILE(3)` = split into 3 buckets

sql

```
SELECT *,  
    NTILE(3) OVER(ORDER BY home_points DESC) AS third  
FROM afc_east  
ORDER BY home_team, game_date;
```

What it does

 Copy code

- Sorts rows by `home_points DESC`
- Divides them into **3 roughly equal-sized groups**
- Labels them `1`, `2`, `3` (bucket numbers)

How this is useful (typical use cases)

- Create "top / middle / bottom" groups
- Quantiles/tertiles for segmentation
- In finance: bucket days into top-third returns vs bottom-third returns, etc.

The big picture (what this file is training you to do)

These patterns map directly to your stock assignment:

- **Per item:** `PARTITION BY ticker`
- **Over time:** `ORDER BY date`
- **YTD average:** add `PARTITION BY ticker, EXTRACT(YEAR FROM date)` + cumulative frame
- **6-day moving average:** `USE ROWS BETWEEN 5 PRECEDING AND CURRENT ROW`

If you paste your stock table name + the "value column" you're using (Close vs Adj Close), I can translate these patterns into the exact window-function skeleton you need (still explaining, not dumping code if you don't want code).

how should i do in R

Thought for 11s >

In R, you'll do the **same "window function" logic** with `dplyr::group_by()` + `arrange()` + `mutate()`:

- **YTD average** \approx cumulative mean within (ticker, year)
- **6-day moving average** \approx rolling mean of the last 6 observations within ticker (ordered by date)

Below is a clean, assignment-ready pattern.

1) Load + clean the time series (Ticker, Date, Value)

Assume your "value" is `close` (you can swap to `Adj Close` if you prefer).

```
r

library(dplyr)
library(readr)
library(lubridate)

df <- read_csv("stocks.csv") %>%
  mutate(Date = as.Date(Date)) %>%
  filter(Date >= as.Date("2022-01-01")) %>%
  arrange(Ticker, Date)
```

Quick sanity checks (worth doing)

```
r

df %>% count(Ticker)
df %>% count(Ticker, Date) %>% filter(n > 1) # duplicates per ticker-date?
```

 Copy code

If you have duplicates, you must resolve them (e.g., keep last, or average).

2) Compute YTD average (resets each year per ticker)

This is the direct R equivalent of:

```
AVG(value) OVER(PARTITION BY ticker, year ORDER BY date ROWS UNBOUNDED PRECEDING)
```

r

```
df2 <- df %>%  
  mutate(year = year(Date)) %>%  
  group_by(Ticker, year) %>%  
  arrange(Date, .by_group = TRUE) %>%  
  mutate(  
    ytd_avg_close = cummean(Close)  
  ) %>%  
  ungroup()
```

✓ `cummean()` is your “running average up to current row”.

 Copy code

3) Compute 6-day moving average (last 6 rows per ticker)

This is the equivalent of:

```
AVG(value) OVER(PARTITION BY ticker ORDER BY date ROWS BETWEEN 5 PRECEDING AND  
CURRENT ROW)
```

Use `slider` (best for this):

r

```
library(slider)  
  
df3 <- df2 %>%  
  group_by(Ticker) %>%  
  arrange(Date, .by_group = TRUE) %>%  
  mutate(  
    ma6_close = slide_dbl(Close, mean, .before = 5, .complete = TRUE)
```

```
) %>%
ungroup()
```

- `.before = 5` + current row = **6 observations**
- `.complete = TRUE` makes the first 5 rows **NA** (because you don't yet have 6 days)

 Copy code

4) Output (what you'll typically show)

```
r

df3 %>%
  select(Ticker, Date, Close, ytd_avg_close, ma6_close) %>%
  arrange(Ticker, Date)
```

ChatGPT 5.2 Thinking ▾

```
r

df3 %>%
  mutate(across(c(ytd_avg_close, ma6_close), ~ round(.x, 2)))
```

 Copy code

Two common "gotchas" (be explicit in writeup)

1. "6-day" means 6 rows, not calendar days.

If your data has missing trading days (weekends/holidays), the rolling window is still 6 observations. That's usually what instructors expect.

2. YTD resets at Jan 1.

That's why we grouped by `Ticker, year(Date)`.

If you paste the **exact column name** you're using for "value" (`Close` vs `Adj Close`) and whether you want the moving average to **reset each year** or **roll continuously**, I'll lock the final version to match your assignment wording exactly.



do i need to keep all the columns?if not ,which col i should keep it(other than tickers and date)

Thought for a second >

No — you **don't** need to keep all columns. For this assignment, you only need enough to compute the two window stats (YTD average + 6-day moving average) **per item over time**.

Minimum columns to keep

You should keep:

- Ticker (item id)
- Date (time)
- **ONE value column** you will average (pick one):
 - Adj Close ☒ *best default for finance* (adjusted for splits/dividends)
 - or Close (ok if you don't care about adjustments)

That's it.

Recommended columns (practical + clean)

Keep: Ticker , Date , Adj Close

Optional (only if you plan to talk about volume or volatility):

- Volume (useful for context like "high volume days")
- High , Low (if you want to mention daily range / volatility)

Columns you can drop (for this assignment)

- Open , High , Low , Close , Volume **if you're not using them**
- Basically: anything you **won't use in calculations or plots**

Why I recommend Adj Close

Because your averages won't get distorted by stock splits/dividends. If you use raw close , a split can create a fake "price drop" that messes up rolling averages.

In R: how you'd trim it

(Example: keep only what you need)

```
r

df_small <- df %>%
  select(Ticker, Date, AdjClose = `Adj Close`)
```

 Copy code

Then compute:

- `ytd_avg = cummean(AdjClose) within group_by(Ticker, year(Date))`
- `ma6 = slide_dbl(AdjClose, mean, .before=5, .complete=TRUE) within group_by(Ticker)`

If your actual file has the column name exactly "Adj Close" (with the space), that's fine — just use backticks like I did above.



do i need to arrange adjclose?

Thought for a couple of seconds >

You don't "arrange **Adj Close**" itself — you **arrange by** `Date` (within each `Ticker`) so the rolling/YTD calculations run in the correct time order.

Why you must sort by Date

Both of these depend on sequence:

- **YTD average** = average from the first day of the year **up to this row** → needs correct chronological order.
- **6-day moving average** = mean of the **current row + previous 5 rows** → needs correct row order.

If the rows aren't sorted by date, your "previous 5 rows" might be random dates → wrong moving average.

What to do in R

Do this before `cummean()` / `slide_*()`:

```
r
```

```
df <- df %>%
  arrange(Ticker, Date)
```

 Copy code

And inside grouped calculations, do:

```
r
```

```
df %>%
  group_by(Ticker) %>%
  arrange(Date, .by_group = TRUE) %>%
  ...
```

When you *don't* need to worry

 Copy code

If you **know** your data is already perfectly sorted by `Ticker`, `Date`, then arranging is technically redundant — but it's still best practice to include it so your results are reproducible and correct even if the file order changes.

So: **No, don't arrange Adj Close. Yes, arrange by Date (per Ticker).**

     ...

in the sample find the difference, why i dont need to do the difference

Thought for 14s >

In `wf_samples.sql`, the “difference” column (`home_differential`) is **just a demo** to show a common window-function pattern:

sql

```
home_points - AVG(home_points) OVER(...) AS home_differential
```

 Copy code

What that “difference” means

It’s a **deviation (residual)**:

- If `home_differential > 0`: this game’s `home_points` is **above** the (overall or team) average
- If `< 0`: **below** average
- If `= 0`: exactly at average

So it answers: “How far is this row from the baseline?”

Why you *don’t* need it for your assignment

Your assignment explicitly asks for **two window-style summaries** per ticker:

1. YTD average
2. 6-day moving average

Both are **averages themselves**, not “difference-from-average” metrics. The sample adds “difference” only to illustrate that once you can compute a window average, you can also subtract it to get deviations—but that subtraction is **not required** unless your rubric says “compare to baseline” or “anomaly detection”.

If you *want* to add it (optional, can make your writeup stronger)

In R you could add:

- difference from YTD average: `close - ytd_avg`
- difference from MA6: `close - ma6`

Example:

r

```
df3 <- df %>%
  arrange(Ticker, Date) %>%
  mutate(year = lubridate::year(Date)) %>%
```

```

group_by(Ticker, year) %>%
  arrange(Date, .by_group = TRUE) %>%
  mutate(ytd_avg = cummean(AdjClose)) %>%
  ungroup() %>%
  group_by(Ticker) %>%
  arrange(Date, .by_group = TRUE) %>%
  mutate(ma6 = slider::slide_dbl(AdjClose, mean, .before = 5, .complete = TRUE))
  ungroup() %>%
  mutate(
    diff_ytd = AdjClose - ytd_avg,
    diff_ma6 = AdjClose - ma6
  )

```

But again: **only do this if you want extra analysis**. For “meet the requirement,” **YTD avg + MA6 is enough**.



Copy code

so i dont need to do the differential?

Thought for a second >

Yes — **you do *not* need to do the differential** *unless* your assignment instructions explicitly say to compute a “difference from average” (or something like “deviation”, “spread”, “excess over baseline”).

For the typical wording you showed earlier, the required outputs are just:

- **YTD average** (per ticker, per year, ordered by date)
- **6-day moving average** (per ticker, ordered by date)

The **differential** in `wf_samples.sql` is a **teaching/example add-on**, not a requirement.

When would you include it?

Only if you want an **extra** column for interpretation, like:

- `AdjClose - ytd_avg` (above/below YTD baseline)
- `AdjClose - ma6` (above/below short-term trend)

But it's optional.

Quick note on your files

I can still see your two SQL files (`wf_samples.sql` , `wf_createtables.sql`). If there were any other uploaded files you want me to reference (like your actual stock CSV), I'm seeing a notice that **some older uploads have expired**—you'd need to upload them again for me to read them.


If you want, paste the exact sentence from the assignment prompt where it lists deliverables, and I'll tell you *exactly* what to output (minimum required vs optional).



week.qmd
File

is it finish?

Thought for 52s >

You're **basically finished**  — your `week.qmd` already does the two required window-style calculations in R:

- **YTD average:** `group_by(Ticker, year) + arrange(date) + cummean(adj_close)`
- **6-day moving average:** `group_by(Ticker) + arrange(date) + slide_dbl(..., .before = 5, .complete = TRUE)`

That said, I would **fix these 4 things before submitting** (they're small but important):

Must-fix (to avoid rubric / grading issues)

1. Prove you meet the “since 2022-01-01” requirement

Right now you *check* min/max date, but you don't **filter**. Add:

- `filter(date >= as.Date("2022-01-01"))`
- and then show `min(date)` is indeed `>= 2022-01-01`.

2. Your min/max date check is using `Date` before converting it

You do:

```
r

stock_raw |> summarise(min_date = min(Date), max_date = max(Date))
```

 Copy code

Better:

```
r

stock_raw |> summarise(
  min_date = min(as.Date(Date), na.rm = TRUE),
  max_date = max(as.Date(Date), na.rm = TRUE)
)
```

3. Remove the typo in your source line

You have:

```
bash

source: "https://.../stocks.csv"3
```

 Copy code

Remove the trailing `3` and format it cleanly.

4. Don't print the entire dataset

You print `stocks` and `stocks_f` fully (could be huge in HTML).

Instead show a small sample, e.g. last few rows per ticker.

 Copy code

Nice-to-have (makes it look “complete”)

- Add 2–3 sentences explaining:
 - YTD avg “resets each year”
 - 6-day moving avg returns **NA for the first 5 rows** because `.complete = TRUE`

One more thing

I'm seeing a notice that **some older uploaded files have expired**. Right now I can review `week.qmd` , but if you want me to verify the *actual* `stocks.csv` date range or values, upload that CSV again (or paste the first/last few rows).

If you want, paste your assignment prompt line about "since 2022-01-01" and I'll tell you whether a **filter is mandatory** or just "preferred."

