Writing SQL Like a Pro: Advanced Techniques Showcased in a Real-Life Scenario

Merging two history tables using advanced SQL techniques with Google's BigQuery.

Furcy Pin · Follow

Published in Learning SQL · 9 min read · Mar 5, 2024

Image by author, generated with Stable Diffusion using the prompt "A pixel art landscape representing advanced SQL techniques, high quality". I've been told that having a nice image is better for SEO, so here it is.

Last week, Arnaud Stephan, a fellow Analytics Engineer, asked for help with an advanced use case that I found very interesting. After spending a couple of hours helping him, I thought it would make a perfect subject for a blog article, and I asked for his permission to discuss this advanced real-life problem in public and publish the associated code.

So, let's saddle up and get ready for some advanced hands-on SQL with my favorite engine: Google BigQuery!

(Note: To preserve anonymity, I replaced the IDs and values with dummy ones.)

About the author:

I am a freelance Analytics Engineer / Data Platform Engineer with more than 12 years of experience in SQL and Big Data. I love to play with complex SQL/DataFrame topics, so if you're stuck on challenging problems that you can't crack with SQL, don't hesitate to reach out; I might be able to help.

The Problem

First, let's give some context: We have data from a grocery retailer selling multiple articles in multiple shops. For each article inside each shop, they track the pricing history of these articles. The corresponding data is contained in the table *sell_price_history,* which looks like this:

```
1  SELECT * FROM `supermarket.sell_price_history` ORDER BY start_datetime
2
```

Query results

JOB INFORMATION | RESULTS | CHART | JSON | EXECUTION DETAILS | EXECUTION GRAPH

Row	article_id	shop_id	sell_price	start_datetime	end_datetime
1	yVxrDmtw	dlr29L9xIF	null	2022-05-10 11:45:19 UTC	2022-05-10 11:53:49 UTC
2	yVxrDmtw	dlr29L9xIF	null	2022-05-10 11:53:49 UTC	2022-05-10 13:08:38 UTC
3	yVxrDmtw	dlr29L9xIF	null	2022-05-10 13:08:38 UTC	2022-05-10 13:08:40 UTC
4	yVxrDmtw	dlr29L9xIF	null	2022-05-10 13:08:40 UTC	2022-05-10 13:08:42 UTC
5	yVxrDmtw	dlr29L9xIF	null	2022-05-10 13:08:42 UTC	2022-05-10 13:08:45 UTC
6	yVxrDmtw	dlr29L9xIF	3.99	2022-05-10 13:08:45 UTC	2022-05-10 13:23:18 UTC
7	yVxrDmtw	dlr29L9xIF	3.99	2022-05-10 13:23:18 UTC	2022-05-11 04:20:13 UTC
8	yVxrDmtw	dlr29L9xIF	3.99	2022-05-11 04:20:13 UTC	2023-06-14 10:46:07 UTC
9	yVxrDmtw	dlr29L9xIF	3.99	2023-06-14 10:46:07 UTC	2024-02-14 00:59:59 UTC

I decided to keep the first rows for this example because this is the kind of weird stuff you get when working with "real" data. The person I helped explained to me that some other columns (that they did not include in the example) did change, so it was important for them to keep those lines, even when the price was null. Image by author.

But each shop can also create *promotions* for any article, in which case the *promotion_value* would replace the original article price. A second table named *promotion_history* has the corresponding data:

```
1  SELECT * FROM `supermarket.promotion_history` ORDER BY start_datetime
2
```

Query results

JOB INFORMATION | RESULTS | CHART | JSON | EXECUTION DETAILS | EXECUTION GRAPH

Row	shop_id	article_id	promotion_id	promotion_value	start_datetime	end_datetime
1	dlr29L9xIF	yVxrDmtw	-38_P0evh	2.99	2022-05-11 00:00:00 UTC	2022-05-11 05:03:04 UTC
2	dlr29L9xIF	yVxrDmtw	-38_P0evh	2.49	2022-05-11 05:03:04 UTC	2022-05-18 00:00:00 UTC
3	dlr29L9xIF	yVxrDmtw	2GLCMItLyw	2.99	2023-06-14 15:05:31 UTC	2023-06-16 11:29:38 UTC
4	dlr29L9xIF	yVxrDmtw	2GLCMItLyw	2.99	2023-06-16 11:29:38 UTC	2023-06-19 08:58:40 UTC
5	dlr29L9xIF	yVxrDmtw	2GLCMItLyw	2.99	2023-06-19 08:58:40 UTC	2023-06-22 13:31:46 UTC
6	dlr29L9xIF	yVxrDmtw	2GLCMItLyw	1.99	2023-06-22 13:31:46 UTC	2023-06-27 23:59:59.999000 U...

Image by author.

The problem then follows: We want to "merge" these two event tables into one single table that will give us, for each article, the history of prices, *including the effects of promotions.*

In other words, the expected result we want should look like this:

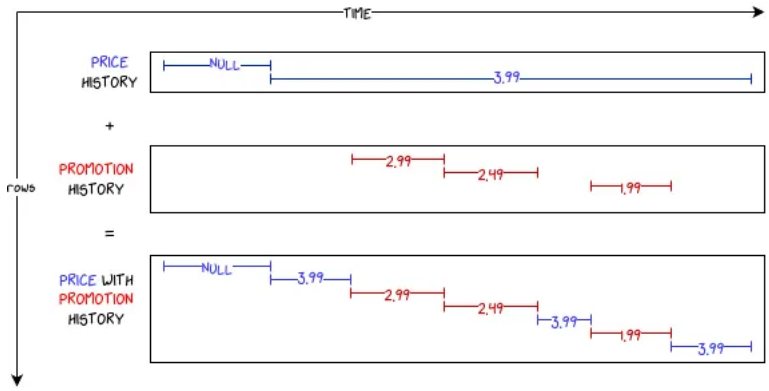
Query results

JOB INFORMATION | RESULTS | CHART | JSON | EXECUTION DETAILS | EXECUTION GRAPH

Row	shop_id	article_id	sell_price	promotion_id	promotion_value	price_with_promotion_included	start_datetime	end_datetime
1	dir29L9xiF	yVxrDmtw	null	null	null	null	2022-05-10 11:45:19 UTC	2022-05-10 11:53:49 UTC
2	dir29L9xiF	yVxrDmtw	null	null	null	null	2022-05-10 11:53:49 UTC	2022-05-10 13:08:38 UTC
3	dir29L9xiF	yVxrDmtw	null	null	null	null	2022-05-10 13:08:38 UTC	2022-05-10 13:08:40 UTC
4	dir29L9xiF	yVxrDmtw	null	null	null	null	2022-05-10 13:08:40 UTC	2022-05-10 13:08:42 UTC
5	dir29L9xiF	yVxrDmtw	null	null	null	null	2022-05-10 13:08:42 UTC	2022-05-10 13:08:45 UTC
6	dir29L9xiF	yVxrDmtw	3.99	null	null	3.99	2022-05-10 13:08:45 UTC	2022-05-10 13:23:18 UTC
7	dir29L9xiF	yVxrDmtw	3.99	null	null	3.99	2022-05-10 13:23:18 UTC	2022-05-11 00:00:00 UTC
8	dir29L9xiF	yVxrDmtw	3.99	-38_PDevh	2.99	2.99	2022-05-11 00:00:00 UTC	2022-05-11 04:20:13 UTC
9	dir29L9xiF	yVxrDmtw	3.99	-38_PDevh	2.99	2.99	2022-05-11 04:20:13 UTC	2022-05-11 05:03:04 UTC
10	dir29L9xiF	yVxrDmtw	3.99	-38_PDevh	2.49	2.49	2022-05-11 05:03:04 UTC	2022-05-18 00:00:00 UTC
11	dir29L9xiF	yVxrDmtw	3.99	null	null	3.99	2022-05-18 00:00:00 UTC	2023-06-14 10:46:07 UTC
12	dir29L9xiF	yVxrDmtw	3.99	null	null	3.99	2023-06-14 10:46:07 UTC	2023-06-14 15:05:31 UTC
13	dir29L9xiF	yVxrDmtw	3.99	2GLCMtLyw	2.99	2.99	2023-06-14 15:05:31 UTC	2023-06-16 11:29:38 UTC
14	dir29L9xiF	yVxrDmtw	3.99	2GLCMtLyw	2.99	2.99	2023-06-16 11:29:38 UTC	2023-06-19 08:58:40 UTC
15	dir29L9xiF	yVxrDmtw	3.99	2GLCMtLyw	2.99	2.99	2023-06-19 08:58:40 UTC	2023-06-22 13:31:46 UTC
16	dir29L9xiF	yVxrDmtw	3.99	2GLCMtLyw	1.99	1.99	2023-06-22 13:31:46 UTC	2023-06-27 23:59:59.999000 U...
17	dir29L9xiF	yVxrDmtw	3.99	null	null	3.99	2023-06-27 23:59:59.999000 U...	2024-02-14 00:59:59 UTC

Image by author.

To give a more visual illustration, we want to merge two tables describing ranges of events.



The problem we want to solve. Image by author.

The (Incorrect) Solution

To make things a bit more lively, I will walk you through how I solved the problem. I had a rough idea that to achieve that, I needed to decompose each table into a kind of "event timeline" and then merge the two timelines and fill in the blanks. This might sound a bit fuzzy for now, but so did it when I started solving the problem. Don't worry; things will get clearer soon.

Step 1: Co-locating the Data

The very first thing I did before even trying to think about how to solve the problem was join the two tables together and group the data by *shop_id* and *article_id,* like this:

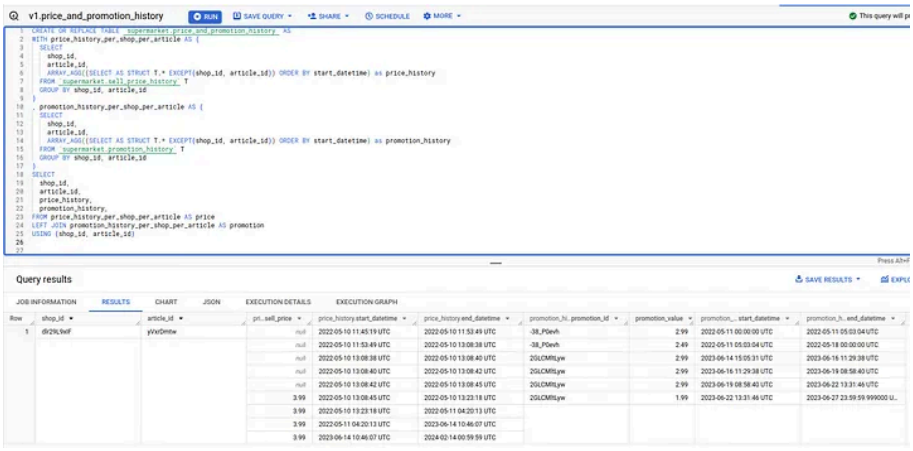


Image by author.

For those of you not familiar with nested structures in BigQuery, this might look a bit weird at first, so here is a detailed explanation of how the data structures are organized:

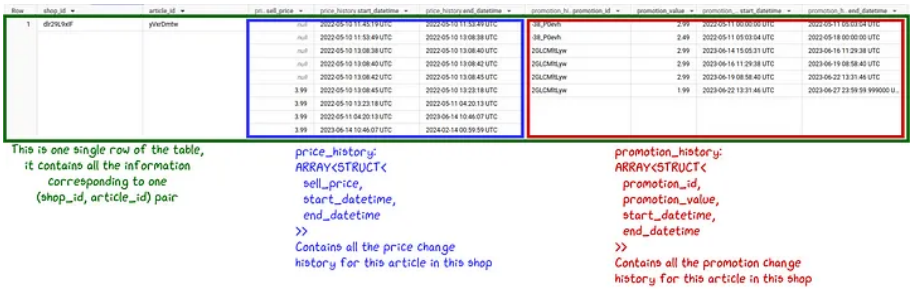


Image by author.

There are multiple reasons for me to prefer this way of arranging things:

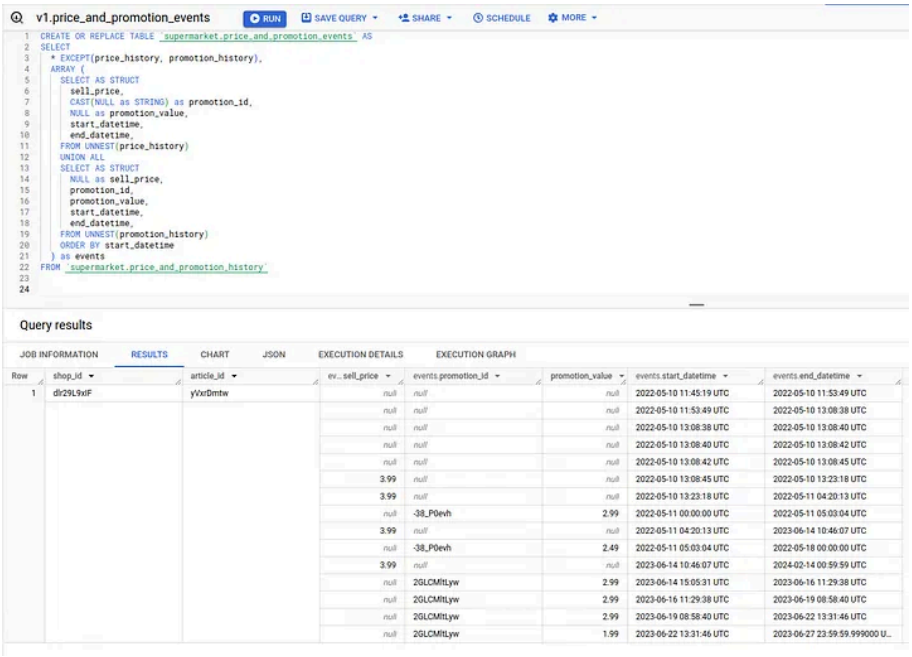
- Simplicity: Co-locating the data for each article allows me to reason on a single-row basis.
- Performance: Thanks to this pre-grouping, all the computations I will perform will be made at the row level. For those familiar with MapReduce, there will only be Map steps, no Reduce, and most of all, no Shuffle, which is generally the most resource-intensive part.
  This is not “crucial” on BigQuery, as query price is only based on the amount of data read and not processed, but it would be on other engines, and I like to optimize my code.
- Storage efficiency: Since *shop_id* and *article_id* are not repeated at each line, this representation takes less storage space for the same quantity of information. This doesn’t make much sense on a toy dataset like this, but to get an idea, the total size of the two input tables was 720b, but the size of the output table (which contains the very same information) was only 412b. Multiply this by billions of lines, and you might notice a price difference.

The main drawbacks are that most people aren't used to it, so they might require some effort to learn how to query it properly, and many metadata tools (data catalogs, data diff, etc.) don't support nested structures very well yet.

But enough digressing. That subject alone would deserve a whole article, so let's go back to the task at hand.

Step 2: Merging the Two Histories

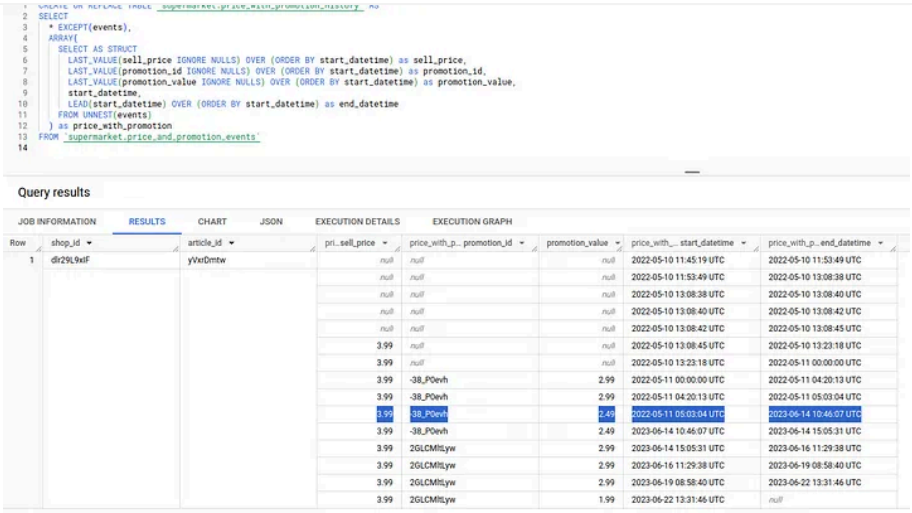
Next, we perform a *union* between the two history tables like this:



To understand this query, imagine that price_history is like a mini-table contained inside the top-level row. Then "SELECT ... FROM UNNEST(price_history)" is a way to query this mini-table while staying inside the top-level row. "SELECT AS STRUCT" is just a way to wrap the result of the select inside a STRUCT. Image by author.

Step 3: Filling in the Blanks

Finally, we use the *last_value* window function to keep the last non-null value in the event orders, to fill in the blanks, or should I say the *NULLs*.

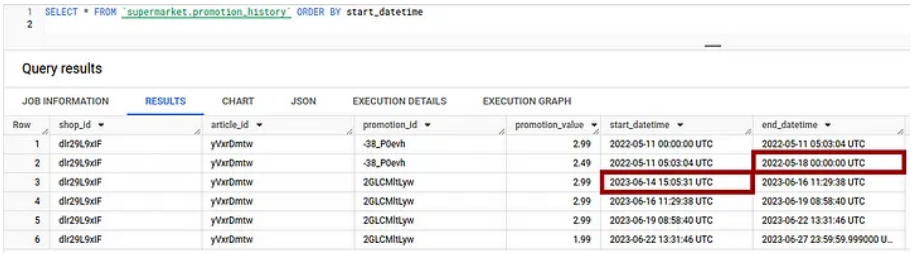


The results look good but aren't. Look closely at the highlighted line. Image by author.

While this result does look like the result we are looking for, sharp readers might already have caught that it is, in fact, incorrect.

Why This Does Not Work:

This would have worked if the promotions went on without any interruption, but if we look more closely at our input data, we see that this is not the case: We do have an interruption between the first and the second promotions. This interruption is not visible in the previous screenshot, as we can see on the line highlighted in blue.



Row	shop_id	article_id	promotion_id	promotion_value	start_datetime	end_datetime
1	dlr29L9xIF	yVxrDmtw	-38_POevh	2.99	2022-05-11 00:00:00 UTC	2022-05-11 05:03:04 UTC
2	dlr29L9xIF	yVxrDmtw	-38_POevh	2.49	2022-05-11 05:03:04 UTC	2022-05-18 00:00:00 UTC
3	dlr29L9xIF	yVxrDmtw	2GLCMltLyw	2.99	2023-06-14 15:05:31 UTC	2023-06-16 11:29:38 UTC
4	dlr29L9xIF	yVxrDmtw	2GLCMltLyw	2.99	2023-06-16 11:29:38 UTC	2023-06-19 08:58:40 UTC
5	dlr29L9xIF	yVxrDmtw	2GLCMltLyw	2.99	2023-06-19 08:58:40 UTC	2023-06-22 13:31:46 UTC
6	dlr29L9xIF	yVxrDmtw	2GLCMltLyw	1.99	2023-06-22 13:31:46 UTC	2023-06-27 23:59:59.999000 U...

The input table "promotion_history" where we highlighted the interruption between promotions. The first promotion ended on 2022–05–18, while the second promotion only started on 2023–06–14, more than one year later. Image by author.

The (Correct) Solution

Now, we will see how the previous (incorrect) solution can be fixed, with more detailed explanations of what is happening.

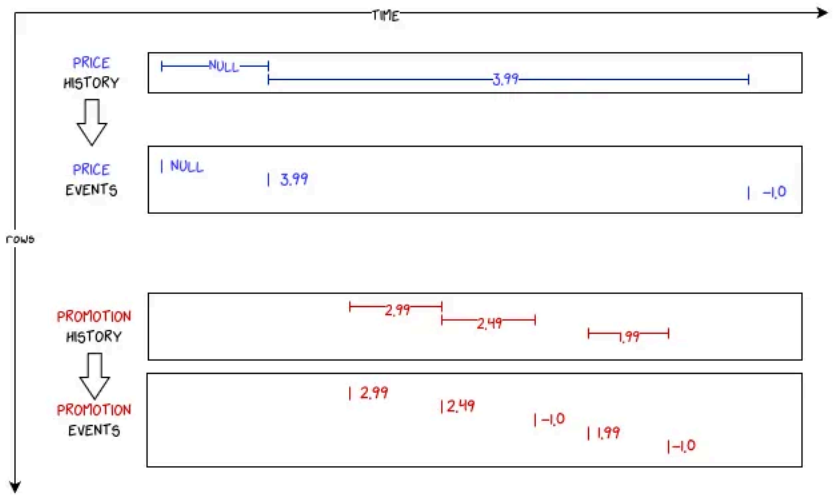
Step 1: Co-locating the Data [Nothing to Change]

Step 1 was just a preparation step. We can keep it as is.

Step 2: Transforming Date Ranges into Timelines and Merging Them

Our intuition of merging the two timelines was good, but we missed the fact that the date ranges could be discontinuous. To fix this, the first step we must achieve is to transform our "history", which is made of date ranges (from *start_date* to *end_date*), into an event timeline, made only of events (just one *event_date*), without losing any information.

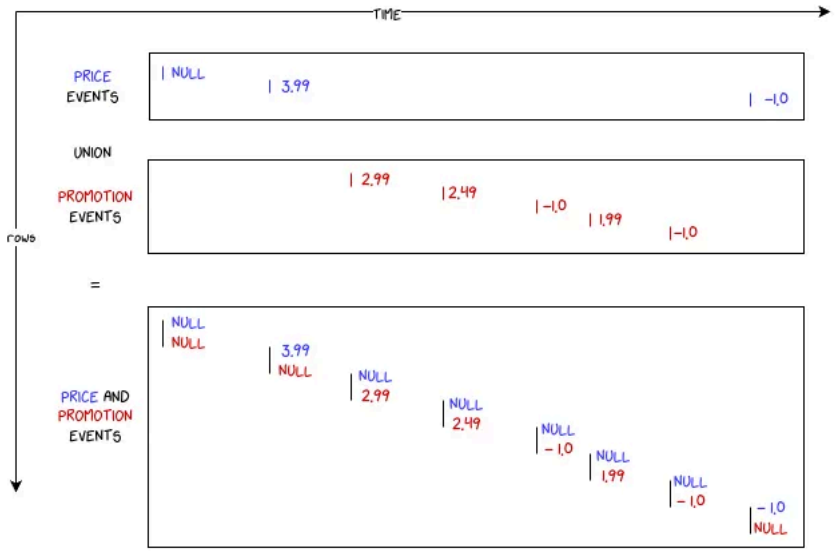
Schematically, this will look like this:



Schematical representation of going from a history (date range) to event representation. Image by author.

Here, the "missing" values have been represented with the *magic number -1*. This is because in step 3. we don't want to treat them the same way as *NULLs* when we use the window function *last_value(... ignore nulls)*. Arguably, this is a code smell, but we will keep it for the sake of simplicity in this article.

Once this is done, we will simply merge the two timelines with a UNION like this:



Schematical representation of unioning two timelines. Image by author.

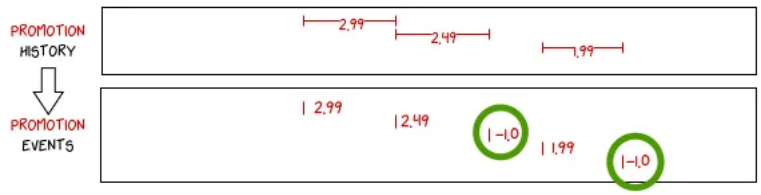
So, what does the code look like? Let's focus only on promotions for a while. The first step is to retrieve the "*start_datetime*" of the next event; this will help us detect when there is a discontinuity:

```
SELECT
  ...
  ARRAY(
    SELECT AS STRUCT
      *,
      LAG(start_datetime) OVER (ORDER BY start_datetime DESC) as next_event_date
    FROM UNNEST(promotion_history)
    ORDER BY start_datetime
  ) as promotion_history,
  ...
```

The second step consists of introducing new events whenever such a discontinuity happens, like this:

```
SELECT
  ...
  ARRAY (
    SELECT AS STRUCT
      promotion_value,
      start_datetime,
    FROM UNNEST(promotion_history)
    UNION ALL
    SELECT AS STRUCT
      -1 as promotion_value,
      end_datetime,
    FROM UNNEST(promotion_history)
    WHERE end_datetime < next_event_datetime OR next_event_datetime IS NULL
  ) as events
  ...
```

The second part of the union is what reintroduces the previously missing "end-events":



Circled in green are the newly added end-events. Image by author.

Bringing everything back together, we get a query in two parts:

```sql
CREATE OR REPLACE TABLE `supermarket.price_and_promotion_history_with_next_event_datetime` AS
SELECT
  shop_id,
  article_id,
  ARRAY(
    SELECT AS STRUCT
      *,
      LAG(start_datetime) OVER (ORDER BY start_datetime DESC) as next_event_datetime
    FROM UNNEST(price_history)
    ORDER BY start_datetime
  ) as price_history,
  ARRAY(
    SELECT AS STRUCT
      *,
      LAG(start_datetime) OVER (ORDER BY start_datetime DESC) as next_event_datetime
    FROM UNNEST(promotion_history)
    ORDER BY start_datetime
  ) as promotion_history,
FROM `supermarket.price_and_promotion_history`
```

Query results

Row	shop_id	article_id	sell_price	price_history.start_datetime	price_history.end_datetime	price... next_event_datetime	promotion_id	promotion_value	promotion_... start_datetime	promotion_h...end_datetime	prom... next_event_datetime
1	dlr29L9xlF	yVxrDmtw	null	2022-05-10 11:45:19 UTC	2022-05-10 11:53:49 UTC	2022-05-10 11:53:49 UTC	-38_P0evh	2.99	2022-05-11 00:00:00 UTC	2022-05-11 05:03:04 UTC	2022-05-11 05:03:04 UTC
			null	2022-05-10 11:53:49 UTC	2022-05-10 13:08:38 UTC	2022-05-10 13:08:38 UTC	-38_P0evh	2.49	2022-05-11 05:03:04 UTC	2022-05-18 00:00:00 UTC	2023-06-14 15:05:31 UTC
			null	2022-05-10 13:08:38 UTC	2022-05-10 13:08:40 UTC	2022-05-10 13:08:40 UTC	2GLCMltLyw	2.99	2023-06-14 15:05:31 UTC	2023-06-16 11:29:38 UTC	2023-06-16 11:29:38 UTC
			null	2022-05-10 13:08:40 UTC	2022-05-10 13:08:42 UTC	2022-05-10 13:08:42 UTC	2GLCMltLyw	2.99	2023-06-16 11:29:38 UTC	2023-06-19 08:58:40 UTC	2023-06-19 08:58:40 UTC
			null	2022-05-10 13:08:42 UTC	2022-05-10 13:08:45 UTC	2022-05-10 13:08:45 UTC	2GLCMltLyw	2.99	2023-06-19 08:58:40 UTC	2023-06-22 13:31:46 UTC	2023-06-22 13:31:46 UTC
			3.99	2022-05-10 13:08:45 UTC	2022-05-10 13:23:18 UTC	2022-05-10 13:23:18 UTC	2GLCMltLyw	1.99	2023-06-22 13:31:46 UTC	2023-06-27 23:59:59.999000 U...	null
			3.99	2022-05-10 13:23:18 UTC	2022-05-11 04:20:13 UTC	2022-05-11 04:20:13 UTC					
			3.99	2022-05-11 04:20:13 UTC	2023-06-14 10:46:07 UTC	2023-06-14 10:46:07 UTC					
			3.99	2023-06-14 10:46:07 UTC	2024-02-14 00:59:59 UTC	null					

Part 1: Retrieving the "start_datetime" of the next event. Image by author.

v2.price_and_promotion_events RUN SAVE QUERY SHARE SCHEDULE MORE

```
CREATE OR REPLACE TABLE `supermarket.price_and_promotion_events` AS
SELECT
  * EXCEPT(price_history, promotion_history),
  ARRAY (
    SELECT AS STRUCT
      sell_price,
      CAST(NULL as STRING) as promotion_id,
      NULL as promotion_value,
      start_datetime,
    FROM UNNEST(price_history)
    UNION ALL
    SELECT AS STRUCT
      -1 as sell_price,
      CAST(NULL as STRING) as promotion_id,
      NULL as promotion_value,
      end_datetime,
    FROM UNNEST(price_history)
    WHERE end_datetime < next_event_datetime OR next_event_datetime IS NULL
    UNION ALL
    SELECT AS STRUCT
      NULL as sell_price,
      promotion_id,
      promotion_value,
      start_datetime,
    FROM UNNEST(promotion_history)
    UNION ALL
    SELECT AS STRUCT
      NULL as sell_price,
      "" as promotion_id,
      -1 as promotion_value,
      end_datetime,
    FROM UNNEST(promotion_history)
    WHERE end_datetime < next_event_datetime OR next_event_datetime IS NULL
    ORDER BY start_datetime
  ) as events,
FROM `supermarket.price_and_promotion_history_with_next_event_datetime`
```

Query results

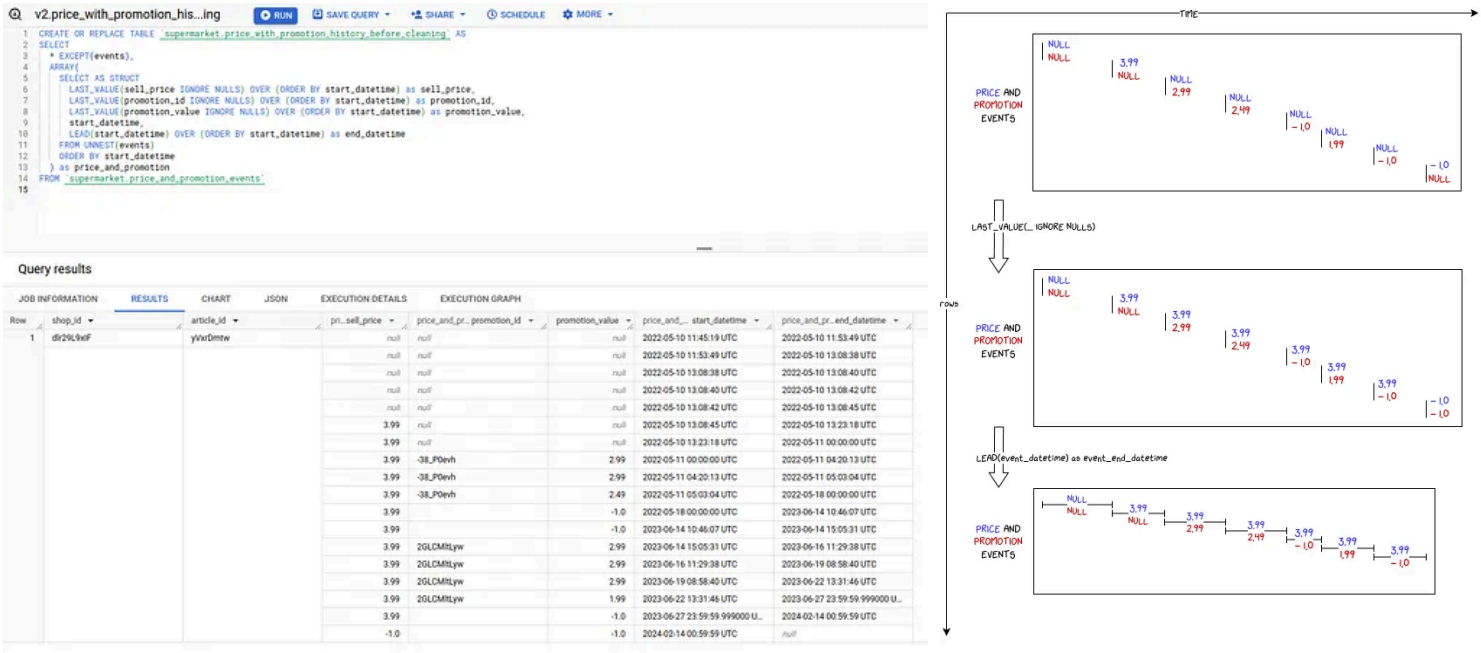
JOB INFORMATION | RESULTS | CHART | JSON | EXECUTION DETAILS | EXECUTION GRAPH

Row	shop_id	article_id	ev... sell_price	events.promotion_id	promotion_value	events.start_datetime
1	dlr29L9xlF	yVxrDmtw	null	null	null	2022-05-10 11:45:19 UTC
			null	null	null	2022-05-10 11:53:49 UTC
			null	null	null	2022-05-10 13:08:38 UTC
			null	null	null	2022-05-10 13:08:40 UTC
			null	null	null	2022-05-10 13:08:42 UTC
			3.99	null	null	2022-05-10 13:08:45 UTC
			3.99	null	null	2022-05-10 13:23:18 UTC
			null	-38_P0evh	2.99	2022-05-11 00:00:00 UTC
			3.99	null	null	2022-05-11 04:20:13 UTC
			null	-38_P0evh	2.49	2022-05-11 05:03:04 UTC
			null		-1.0	2022-05-18 00:00:00 UTC
			3.99	null	null	2023-06-14 10:46:07 UTC
			null	2GLCMltLyw	2.99	2023-06-14 15:05:31 UTC
			null	2GLCMltLyw	2.99	2023-06-16 11:29:38 UTC
			null	2GLCMltLyw	2.99	2023-06-19 08:58:40 UTC
			null	2GLCMltLyw	1.99	2023-06-22 13:31:46 UTC
			null		-1.0	2023-06-27 23:59:59.999000 U...
			-1.0	null	null	2024-02-14 00:59:59 UTC

Part 2: Adding the end-event and merging the two timelines. We do both in one go which is why we are unioning 4 sub-tables together. Note that we also used a magic value “” for the promotion_id. Image by author.

Step 3: Filling in the Blanks + Replacing Magic Values

Now that this is done, the previous query from step 3 gives us a result that looks almost like the one we want:



This is the same query as in our first (incorrect) solution. But the inputs are different, so the output looks much better.
Image by author.

The window function *last_value(… ignore nulls)* allows us to fill in the blanks, and the window function *lead* lets us retrieve the *end_dates* to re-form our date ranges.

All we need to do next is perform some cleaning to remove the magic values *-1* and *“”*:

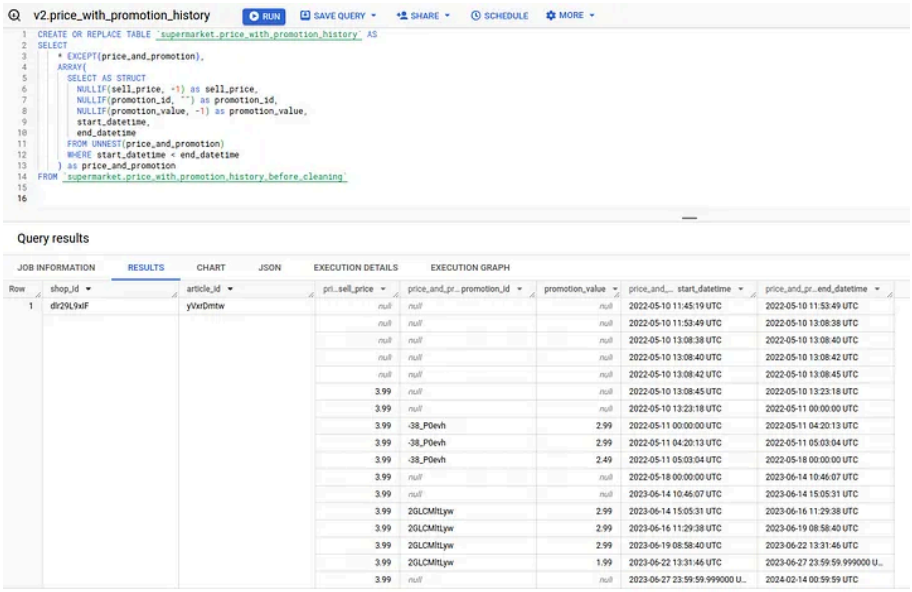
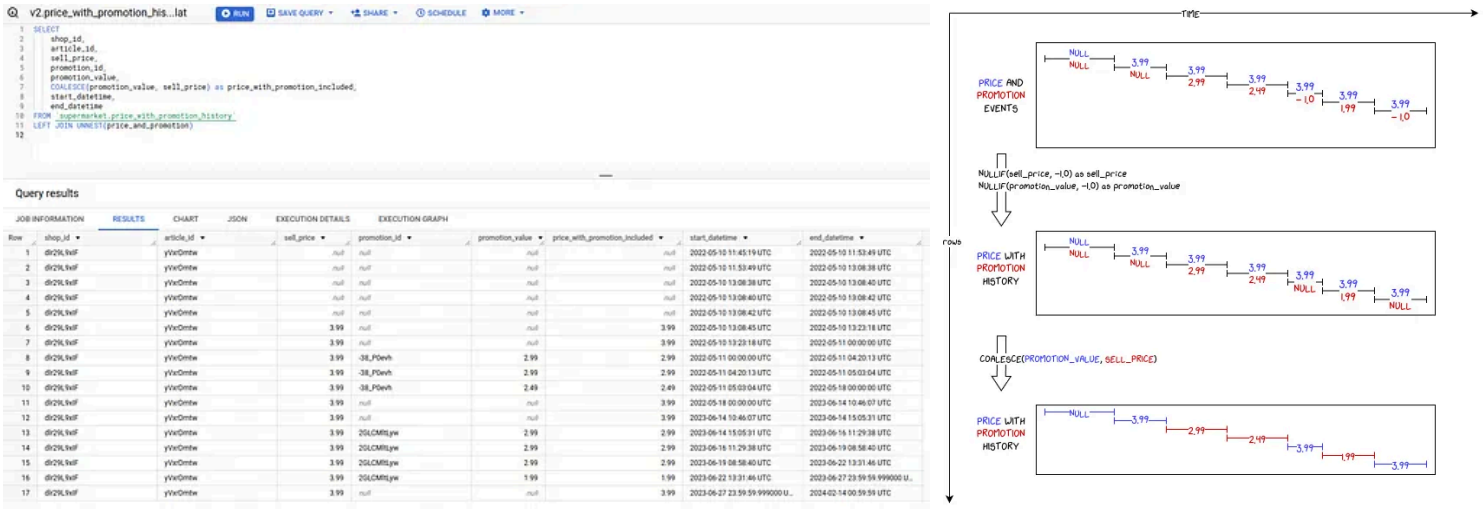


Image by author.

We perform one final query to compute the *price_with_promotion_included* and one final *unnest* since we want the final results to be flat (*shop_id* and

article_id are repeated at every row).



Removing the magic values with NULLIF and performing one final COALESCE to compute the price_with_promotion_included. Images by author.

And that's it, we are done!

Conclusion

We managed to compute what we wanted in four or five steps. I think this problem is a good example of something that looks simple at first sight but is rather tricky to do with SQL. This is the kind of advanced transformation where I recommend considering the trade-off between writing the logic in pure SQL vs. writing a User Defined Function (UDF) in another language (Python, Scala, Javascript, or whatever), if possible. The UDF has the advantage of making the code more generic and safe (with unit tests). However, it has the drawback of being generally less efficient and more costly compared to pure SQL.

As an exercise, if you want some practice, I recommend trying to rewrite this exercise yourself with one of the following variants:

- Rewrite this without using magic values.
  (Hint: Add an *event_status* column that can take the value "*ON*" or "*OFF*".)
- Rewrite this without using *ARRAYS* or *STRUCT*.
  (Hint: It is quite doable, but the windows might need an extra "*PARTITION BY shop_id, article_id".)*
- Rewrite it using your favorite dbt-like tool.
  (I voluntarily did not use dbt here to keep this blog post accessible to most.)

- Try to write this in a fully generic way: The query should work with any number of columns in the input tables and keep working even if we add a new column in the input table without updating the query.
  (Hint: I'm not sure this one is possible in pure-SQL because when I tried, I noticed that BigQuery seems unable to automatically cast NULLs into complex types when performing a union. This might be a nice challenge to do with bigquery-dataframes or bigquery-frames, though).
- With DataFrames, it's probably even possible to make a fully generic transformation that would work automatically with any number of history tables.

Finally, if you want to learn more about how to work with arrays, you can start with this awesome page from BigQuery's documentation, and if you would like to see more articles on that specific subject from me, please leave a comment to let me know.

That's all for today. Thank you for reading! I hope you enjoyed reading this as much as I did writing it.

I leave you with the full query used in this article (rewritten with CTEs) and the full explanation graph. The code and the data samples are available here. Enjoy ;-)

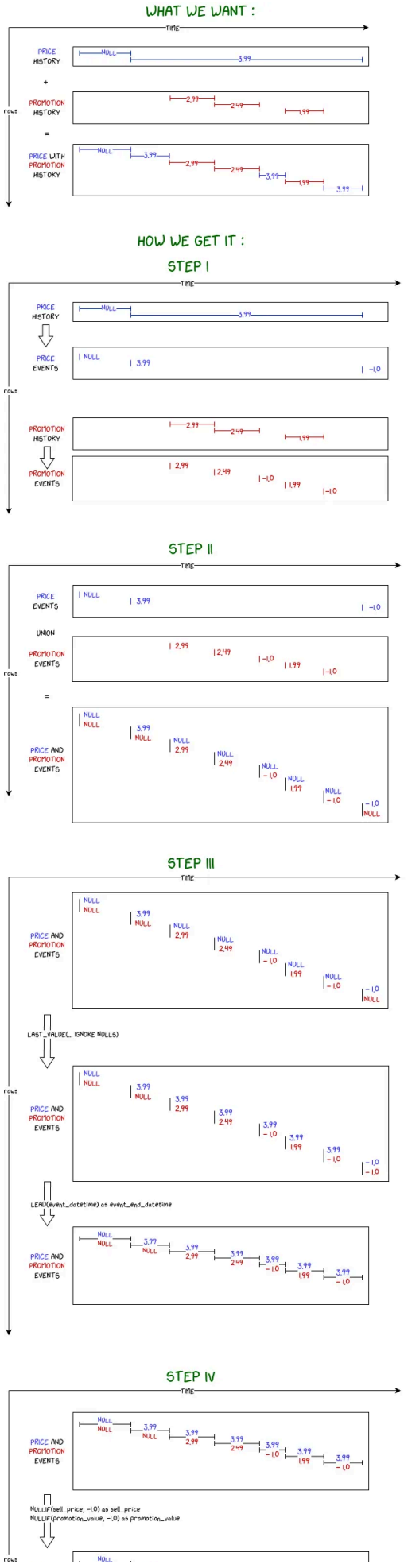
Links

Here are all the links shared in this article:

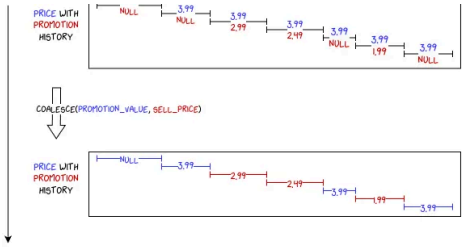
- python-bigquery-dataframes: Google's official project of Python-based DataFrames that runs on BigQuery.
- bigquery-frames: My very own project of Python-based DataFrames that runs on BigQuery.
- Google BigQuery's documentation: Working with arrays.
- Git gist with this article's code.

v2.full RUN SAVE QUERY SHARE SCHEDULE MORE

```sql
WITH price_history_per_shop_per_article AS (
  SELECT
    shop_id,
    article_id,
    ARRAY_AGG((SELECT AS STRUCT T.* EXCEPT(shop_id, article_id)) ORDER BY start_datetime) as price_history
  FROM `supermarket.sell_price_history` T
  GROUP BY shop_id, article_id
)
, promotion_history_per_shop_per_article AS (
  SELECT
    shop_id,
    article_id,
    ARRAY_AGG((SELECT AS STRUCT T.* EXCEPT(shop_id, article_id)) ORDER BY start_datetime) as promotion_history
  FROM `supermarket.promotion_history` T
  GROUP BY shop_id, article_id
)
, price_and_promotion_history AS (
  SELECT
    shop_id,
    article_id,
    price_history,
    promotion_history,
  FROM price_history_per_shop_per_article AS price
  LEFT JOIN promotion_history_per_shop_per_article AS promotion
  USING (shop_id, article_id)
)
, price_and_promotion_history_with_next_event_datetime AS (
  SELECT
    shop_id,
    article_id,
    ARRAY(
      SELECT AS STRUCT
        *,
        LAG(start_datetime) OVER (ORDER BY start_datetime DESC) as next_event_datetime
      FROM UNNEST(price_history)
      ORDER BY start_datetime
    ) as price_history,
    ARRAY(
      SELECT AS STRUCT
        *,
        LAG(start_datetime) OVER (ORDER BY start_datetime DESC) as next_event_datetime
      FROM UNNEST(promotion_history)
      ORDER BY start_datetime
    ) as promotion_history,
  FROM price_and_promotion_history
)
, price_and_promotion_events AS (
  SELECT
    * EXCEPT(price_history, promotion_history),
    ARRAY (
      SELECT AS STRUCT
        sell_price,
        CAST(NULL as STRING) as promotion_id,
        NULL as promotion_value,
        start_datetime,
      FROM UNNEST(price_history)
      UNION ALL
      SELECT AS STRUCT
        -1 as sell_price,
        CAST(NULL as STRING) as promotion_id,
        NULL as promotion_value,
        end_datetime,
      FROM UNNEST(price_history)
      WHERE end_datetime < next_event_datetime OR next_event_datetime IS NULL
      UNION ALL
      SELECT AS STRUCT
        NULL as sell_price,
        promotion_id,
        promotion_value,
        start_datetime,
      FROM UNNEST(promotion_history)
      UNION ALL
      SELECT AS STRUCT
        NULL as sell_price,
        "" as promotion_id,
        -1 as promotion_value,
        end_datetime,
      FROM UNNEST(promotion_history)
      WHERE end_datetime < next_event_datetime OR next_event_datetime IS NULL
      ORDER BY start_datetime
    ) as events,
  FROM price_and_promotion_history_with_next_event_datetime
)
, price_with_promotion_history_before_cleaning AS (
  SELECT
    * EXCEPT(events),
    ARRAY(
      SELECT AS STRUCT
        LAST_VALUE(sell_price IGNORE NULLS) OVER (ORDER BY start_datetime) as sell_price,
        LAST_VALUE(promotion_id IGNORE NULLS) OVER (ORDER BY start_datetime) as promotion_id,
        LAST_VALUE(promotion_value IGNORE NULLS) OVER (ORDER BY start_datetime) as promotion_value,
        start_datetime,
        LEAD(start_datetime) OVER (ORDER BY start_datetime) as end_datetime
      FROM UNNEST(events)
      ORDER BY start_datetime
    ) as price_and_promotion
  FROM price_and_promotion_events
)
, price_with_promotion_history AS (
  SELECT
    * EXCEPT(price_and_promotion),
    ARRAY(
      SELECT AS STRUCT
        NULLIF(sell_price, -1) as sell_price,
        NULLIF(promotion_id, "") as promotion_id,
        NULLIF(promotion_value, -1) as promotion_value,
        start_datetime,
        end_datetime
      FROM UNNEST(price_and_promotion)
      WHERE start_datetime < end_datetime
    ) as price_and_promotion
```



```
112   FROM price_with_promotion_history_before_cleaning
113   )
114   SELECT
115       shop_id,
116       article_id,
117       sell_price,
118       promotion_id,
119       promotion_value,
120       COALESCE(promotion_value, sell_price) as price_with_promotion_included,
121       start_datetime,
122       end_datetime
123   FROM price_with_promotion_history
124   LEFT JOIN UNNEST(price_and_promotion)
125
```



The full query is 124 lines long. Images by author.

Analytics Engineering | Bigquery | Sql | Data Engineering | Editors Pick

More from the list: "SQL"

Curated by @reenum

SQL Fundam... in Stackad...
9 Advanced SQL Queries for Data Mastery
3 min read · Jan 20, 2024

SQL Fundam... in Stackad...
Top 15 Advanced SQL Queries
5 min read · Feb 24, 2024

SQL Fundam... in DevOps...
Top 10 Advanced SQL Queries
3 min read · Nov 7, 2023

Les SQ
11 m

View list

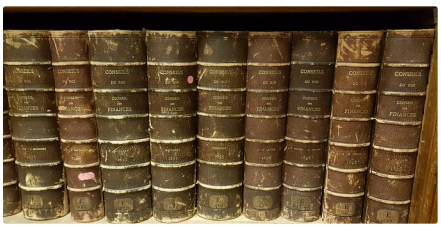
Written by Furcy Pin

Follow

871 Followers · Writer for Learning SQL

[Available for freelance work] Data Engineer, Data Plumber, Data Librarian, Data Smithy.

More from Furcy Pin and Learning SQL



Furcy Pin in Towards Data Science

2003–2023: A Brief History of Big Data

Summing up 20 years of history of Hadoop and everything related

18 min read · Nov 9, 2022

342 4

Sarang S. Babu in Learning SQL

12 Tips for Optimizing SQL Queries for Faster Performance

Ways to Optimize SQL Queries

8 min read · Mar 6, 2023

120 2

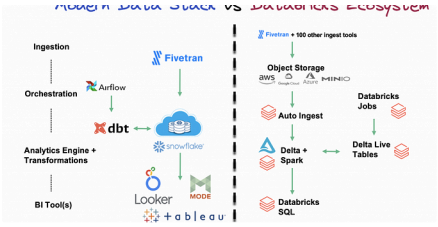
Zach Quinn in Learning SQL

How I Reduced My Query's Run Time From 30 Min. To 30 Sec. In 1...

The query optimization steps a senior data engineer took to reduce the process time of ...

· 6 min read · Mar 13, 2024

413 16



Furcy Pin in Towards Data Science

Modern Data Stack: which place for Spark ?

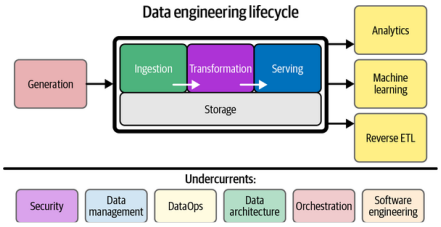
One year ago, some were already predicting that dbt will one day become bigger than...

7 min read · Jan 25, 2022

692 15

See all from Furcy Pin | See all from Learning SQL

Recommended from Medium



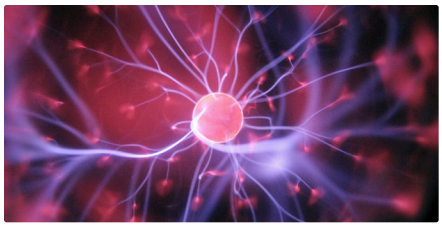
Mudra Patel

Data Engineering concepts: Part 1, Data Modeling

What is Data Engineering?

5 min read · Feb 11, 2024

332 3



Rosaria Silipo in Low Code for Data Science

Is Data Science dead?

In the last six months I have heard this question thousands of time: “Is data science...

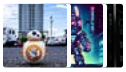
6 min read · Mar 11, 2024

1.7K 36

Lists

The New Chatbots: ChatGPT, Bard, and Beyond
12 stories · 360 saves

ChatGPT
21 stories · 586 saves



Natural Language Processing
1386 stories · 882 saves

Business
41 stories · 96 saves

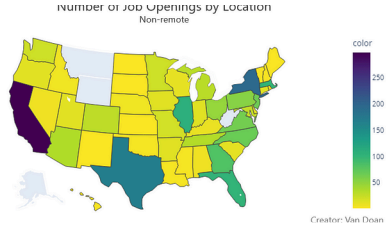
Kallol Mazumdar in ILLUMINATION

I Went on the Dark Web and Instantly Regretted It

Accessing the forbidden parts of the World Wide Web, only to realize the depravity of...

8 min read · Mar 13, 2024

15.3K 302



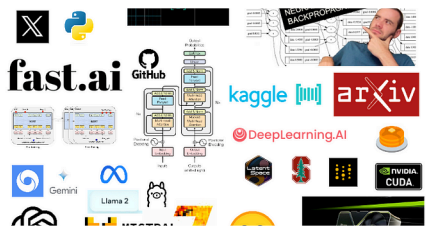
Luna (Van) Doan

Data Job Market 2024: Insights You Need to Boost Your Career

Level up your data career with the insights from 3K+ LinkedIn jobs posts

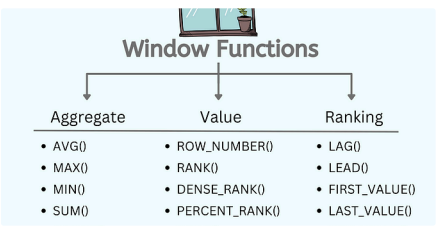
5 min read · Feb 19, 2024

1.1K 21



Benedict Neo in bitgrit Data Science Publication

Roadmap to Learn AI in 2024



Hasan Hüseyin Coşgun

A free curriculum for hackers and programmers to learn AI

11 min read · Mar 10, 2024

10.3K 111

Learn SQL Window Functions With Queries

Time to discover the world of Window Functions with examples; lead( ), lag( ),...

7 min read · Nov 3, 2023

88

See more recommendations