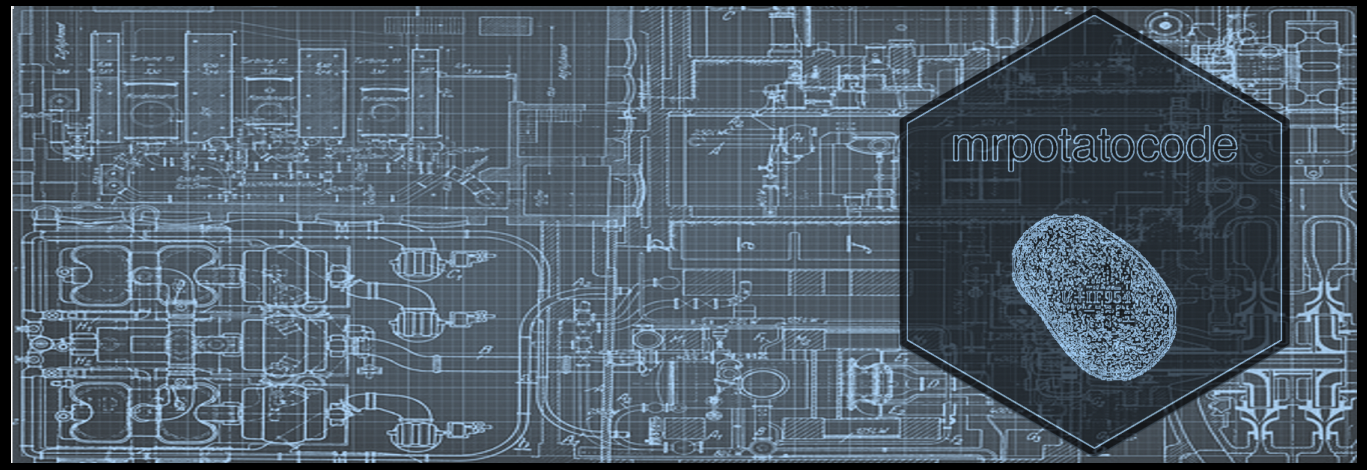


Introduction to Data Access and Storage

Thomas Rosenthal - DSI @ UofT

Module 04

Advanced Techniques:

Windowed Functions

String Manipulation

CROSS JOIN

Self Joins

UNION & UNION ALL

INTERSECT & EXCEPT

Advanced Techniques:

Windowed Functions

String Manipulation

CROSS JOIN

Self Joins

UNION & UNION ALL

INTERSECT & EXCEPT

Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

Purpose

- Windowed Functions allow us to create groupings within groupings ("partitions")
- Allow for greater complexity than simple SQL
  - In Module 3, we mentioned briefly a rolling total, e.g. a `SUM` and a `COUNT`; windowed functions allow us to return these types of results

- Often used with a subquery
  - One of the most common techniques is creating a row number `ROW_NUMBER()` per group
    - When combined with `ORDER BY`, the associated row number will be the *highest* or *lowest* per grouping
    - This allows you to select the min or max by setting the row number = 1 in the "outer" query (i.e. not the "inner" subquery)

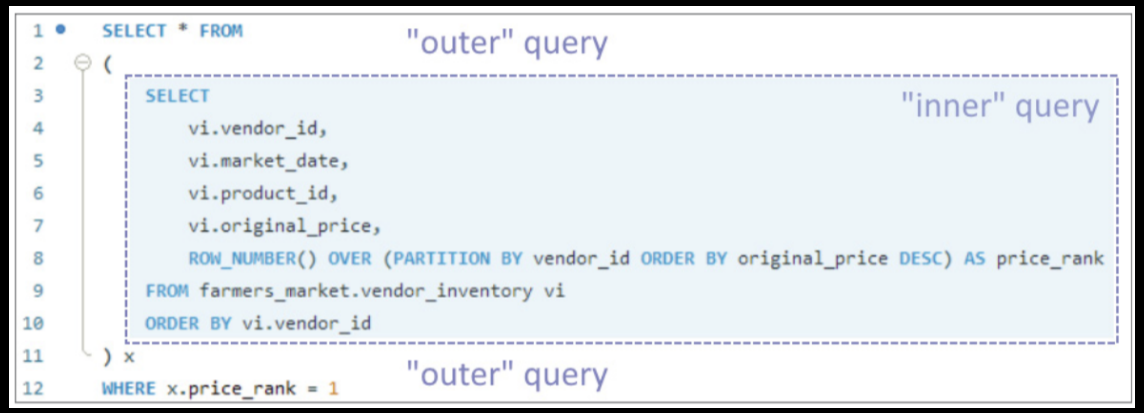


Image: Teate, Chapter 7

Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

OVER

- Syntax for windowed function always requires the `OVER` clause
  - `{desired_windowed_function} OVER (ORDER BY [a column])`
- The `ORDER BY` clause is required
- Think of the `OVER` clause as applying the function of your choice
  - e.g. create row numbers based on the ordering of this column
  - e.g. rank these values from highest to lowest

Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

PARTITION BY

- Within an `OVER` clause, we can optionally use `PARTITION BY` to create groupings for the function to be applied to
  - `{desired_windowed_function} OVER (PARTITION BY [a column] ORDER BY [a column])`
- Now, the function is being applied to different groups
  - e.g. rank these values from highest to lowest within these groups
    - The ranking will restart for each group
    - Think of this like the Olympics: the top three competitors for each event get gold, silver, and bronze — the `PARTITION BY` is the event, the `ORDER BY` is the time `ASC` or points `DESC` that determine the outcome of the event
- Both the `PARTITION BY` and `ORDER BY` arguments can take more than one column
  - e.g. life expectancy by country by continent

Windowed Functions

Purpose

OVER

PARTITION BY

ROW_NUMBER()

Other Windowed Functions

ROW_NUMBER()

- `ROW_NUMBER()` is the simplest windowed function, but also one of the most useful
  - There are no mathematical functions being applied, just an incremental value by group
  - Determining the top (or bottom) per group is often done through `ROW_NUMBER()`

- `ROW_NUMBER()` might feel a bit like ranking `RANK()`...but it's not quite
  - What is the difference between `ROW_NUMBER()` and `RANK()`?

Windowed Functions

Purpose

OVER

PARTITION BY

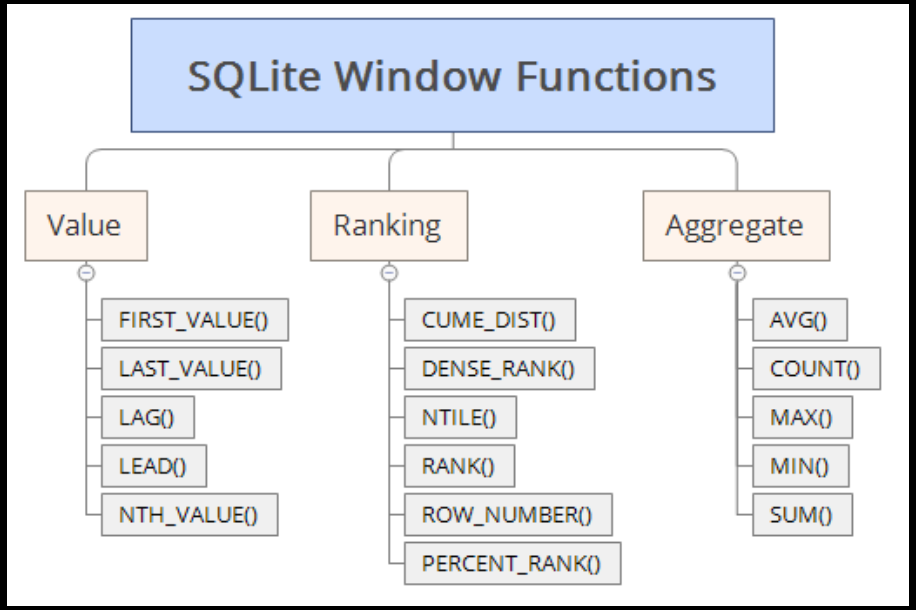
ROW_NUMBER

Other Windowed Functions

Other Windowed Functions

- SQL supports quite a few other windowed functions
- `NTILE` for example will assign rows to buckets (4: quartile, 5: quintile, 10: decile, etc)
  - As such, the `NTILE` function requires an argument passed to it
    - `NTILE(4) OVER (PARTITON BY...ORDER BY...)`

- `LAG` and `LEAD` allow us to create an offset of another column
  - e.g. show a previous_year_total next to a current_year_total for easy comparison
- Knowing how and why to use these can make querying a lot easier



Windowed Functions

(Windowed Functions live coding)

Windowed functions are hard! Questions about them?

Advanced Techniques:

Windowed Functions

String Manipulation

CROSS JOIN

Self Joins

UNION & UNION ALL

INTERSECT & EXCEPT

String Manipulation

LTRIM & RTRIM

REPLACE

UPPER & LOWER

Concatenation

(…)

String Manipulation (continued)

SUBSTR

INSTR

LENGTH

CHAR & UNICODE

REGEXP

String Manipulation

LTRIM & RTRIM

REPLACE

UPPER & LOWER

Concatenation

(...)

LTRIM & RTRIM

- `LTRIM` and `RTRIM` serve two purposes in SQLite:
  - Their main function is to remove leading or trailing white spaces from strings
    - This is surprisingly common — many SQL databases are populated by human input, and this is a frequently overlooked input error
    - e.g. ' Thomas Rosenthal '
  - Alternatively, they act similarly to `REPLACE` (coming up next), but within their specific context:
    - `LTRIM` removes any specified set of characters from the *left*
    - `RTRIM` removes any specified set of characters from the *right*
      - The usefulness of this is going to be very case specific:
        - e.g. wanting to remove a prefix/suffix of an ID:
          - `LTRIM("A189A", 'A')` would result in '189A'
          - `RTRIM("A189A", 'A')` would result in 'A189'
          - `REPLACE` would remove both A's: '189'

LTRIM & RTRIM

(`LTRIM` & `RTRIM` live coding)

String Manipulation

LTRIM & RTRIM

REPLACE

UPPER & LOWER

Concatenation

(...)

REPLACE

- `REPLACE` is likely going to be one of your most commonly used string manipulations
- It substitutes a character or set of characters with another
  - We specify which string (or set of strings within a column), what we want to replace, and the replacement value
    - e.g. `REPLACE('A is an excellent instructor','instructor','TA')` results in 'A is an excellent TA'
  - You can also replace a character with nothing, using an empty string: `''`
    - e.g. `REPLACE('colour','u','')` results in 'color'

- `REPLACE` statements can be strung together — the innermost function will be executed first
  - e.g. `REPLACE(REPLACE(REPLACE(REPLACE('A?lot-of,punctuation.','.',' '),',',' '),'-',' '),'?',' ')` results in 'A lot of punctuation'

REPLACE

(REPLACE live coding)

String Manipulation

LTRIM & RTRIM

REPLACE

UPPER & LOWER

Concatenation

(...)

UPPER & LOWER

- `UPPER` forces all string characters to be uppercase
- `LOWER` forces all string characters to be lowercase
- Both `UPPER` and `LOWER` are essential for filtering tables based on strings
  - It's always best to assume that there is some string variety
  - Sometimes a `LIKE` statement will not be an option

annoying_string_column
WORD
Word
word
wOrD
DifferentWord

- We can always use `UPPER` or `LOWER` in a `WHERE` clause, even without using the commands in the `SELECT` statement

```
SELECT annoying_string_column
FROM table
WHERE LOWER(annoying_string_column) = 'word'
```

- *(This is also true for all of these string manipulations!)*

UPPER & LOWER

(`UPPER` & `LOWER` live coding)

String Manipulation

LTRIM & RTRIM

REPLACE

UPPER & LOWER

Concatenation

(...)

Concatenation (sometimes CONCAT, flavour dependent)

- String concatenation combines two or more columns into a single column
- Concatenation can handle non-column values too
  - e.g. `first_name || ' ' || last_name as full_name`
  - Or `last_name || ', ' || first_name AS full_name`
- In SQLite, `CONCAT` is replaced by two vertical bar characters: `||`
  - Most other flavours use `CONCAT`
- By default, spaces are not included between columns
  - i.e. you need to add a blank space between quotes

Concatenation

(Concatenation live coding)

String Manipulation (continued)

SUBSTR

INSTR

LENGTH

CHAR & UNICODE

REGEXP

SUBSTR ("substring")

- `SUBSTR` specifies any section of a string to return, based on:
  - Which string (i.e. column)
  - Where to begin the section (i.e. the string position to start, as an integer)
  - The (optional) number of characters to return (i.e. how far to go, as an integer)

- `SUBSTR` replaces flavour specific functions like `LEFT` or `RIGHT`
  - By default `SUBSTR` counts from the left
    - e.g. `substr('a long string', 3, 4)` will return "long"
  - To count from the right, specify a negative number to start
    - e.g. `substr('a long string', -6, 6)` will return "string"

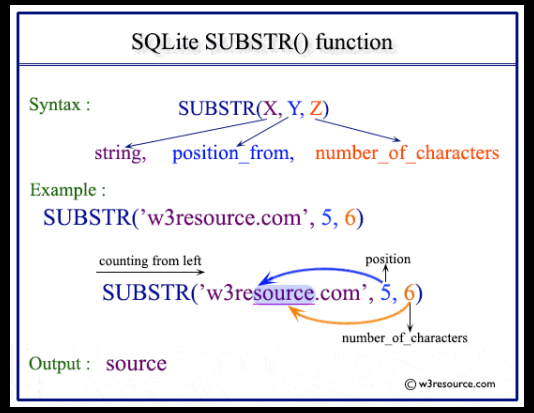


Image: www.w3resource.com/sqlite/core-functions-substr.php

SUBSTR

(SUBSTR live coding)

String Manipulation (continued)

SUBSTR

INSTR

LENGTH

CHAR & UNICODE

REGEXP

INSTR (CHARINDEX flavour dependent)

- `INSTR` provides the starting position or location of a specified string
- `INSTR('The instructor is named Thomas','Thomas')` will result in 25, because "Thomas" is the 25th through 30th character in our string
  - `INSTR('ihe Instructor is named Thomas','Th')` will result in 1 because "Th" arises in "The" before "Thomas"

- `INSTR` can help with splitting a text string on delimiters
  - By finding the distance between delimiters and extracting the appropriate characters with `SUBSTR` we can move through delimiters in text columns
  - The code get's a wild quite quickly:

```
SELECT
 SUBSTR('FirstWord, SecondWord, ThirdWord',0, INSTR('FirstWord, SecondWord, ThirdWord',',')) as FirstDelim
,SUBSTR('FirstWord, SecondWord, ThirdWord',
  INSTR('FirstWord, SecondWord, ThirdWord',',')+1,
  INSTR('FirstWord, SecondWord, ThirdWord',',')+1) as SecondDelim
,SUBSTR('FirstWord, SecondWord, ThirdWord',
  INSTR(
    (SUBSTR('FirstWord, SecondWord, ThirdWord',
    INSTR('FirstWord, SecondWord, ThirdWord',',')+1))
  ,',') +
  INSTR('FirstWord, SecondWord, ThirdWord',',')+1) AS ThirdDelim
```

INSTR

(`INSTR` live coding)

String Manipulation (continued)

SUBSTR

INSTR

LENGTH

CHAR & UNICODE

REGEXP

LENGTH

- `LENGTH` returns the number of characters in a given string (or set of strings in a column)
  - `LENGTH` also works on integers
- `LENGTH` is perhaps less of a string manipulation in and of itself, but is useful in debugging
  - Combined with `MAX`, `LENGTH` can be useful, especially when adding string length constraints to a column
  - Combined with `SUBSTR`, `LENGTH` can cut strings within a column by a dynamic value
- What happens when we apply `SELECT SUBSTR(CanadianMusicians, 0, LENGTH(CanadianMusicians)-6)` to the table below?

CanadianMusicians
Neil Young
Leonard Cohen
Shania Twain
Michael Bublé

CanadianMusicians
Neil
Leonard
Shania
Michael

LENGTH

(`LENGTH` live coding)

String Manipulation (continued)

SUBSTR

INSTR

LENGTH

CHAR & UNICODE

REGEXP

CHAR & UNICODE

CHAR

- When provided an ASCII value, `CHAR` will return the appropriate character from the ASCII table
  - e.g. `CHAR(98)` will result in 'b'
- Pronunciation is split on "char":
  - "char" as in "*char*-broiled"
  - "char" as in "*car*"
  - "char" as in "*char*acter"
  - "char" as in "*care*"

- `CHAR` is hugely useful with `REPLACE`
  - Occasionally, line breaks affect SQL column validity, so `REPLACE(lf_column,CHAR(10),'')` and/or `REPLACE(cr_column,CHAR(13),'')` will be hugely useful
    - Where `CHAR(10)` is a linefeed "lf" and `CHAR(13)` is a carriage return "cr"
- `CHAR` can help with structure and control of strings as they flow into columns

CHAR & UNICODE

UNICODE (ASCII in some flavours)

- `UNICODE` provides the ASCII value of any given character
  - i.e. the opposite of `CHAR`
- The usage? I'm a bit unsure! Maybe faster than looking it up online?
  - e.g. `UNICODE('b')` will result in '98'

CHAR & UNICODE

(`CHAR` & `UNICODE` live coding)

String Manipulation (continued)

SUBSTR

INSTR

LENGTH

CHAR & UNICODE

REGEXP

REGEXP (flavour dependent)

- `REGEXP` allows for string filtering based on regular expressions (regex)
- Situated within a `WHERE` clause, very similar to `LIKE`

- Can use either SQL's or regex's Boolean operators
  - e.g. `WHERE austen_books REGEXP '(sion|ice)$'`
  - Or `WHERE austen_books REGEXP 'sion$' OR book_title REGEXP 'ice$'`

austen_books
Sense & Sensibility
Pride & Prejudice
Mansfield Park
Emma
Persuasion
Northanger Abbey

austen_books
Pride & Prejudice
Persuasion

REGEXP (flavour dependent)

(Quick `REGEXP` live coding)

> Some people, when confronted with a problem think: "I know, I'll use regular expressions." Now they have two problems.
>
> Jamie Zawinski (probably)

LOWER(qUeStIoNs?)

Advanced Techniques:

String Manipulation

CROSS JOIN

Self Joins

UNION & UNION ALL

INTERSECT & EXCEPT

CROSS JOIN

- `CROSS JOIN` creates an unfiltered Cartesian Product
- They are not joined on any columns

- Recall our deck of cards example in Module 2:

```
SELECT suit, rank
FROM suits
CROSS JOIN ranks
```

- Because tables 'suits' and 'ranks' contain no common columns, we would have no other means to join

- I love to `CROSS JOIN`!
- They can be super useful when used correctly
  - What are some good examples that could be useful?

CROSS JOIN

(`CROSS JOIN` live coding)

> No complex query is complete without at least one `CROSS JOIN`
>
> (me, jokingly)

CROSS JOIN Questions?

Advanced Techniques:

String Manipulation

CROSS JOIN

Self Joins

UNION & UNION ALL

INTERSECT & EXCEPT

Self Joins

- Self Joins are somewhat uncommon, but are the last type of possible join
- They are useful for comparison:
    - Determine maximum to-date
    - Generating pairings
- They can help with hierarchy
    - Child-to-Parent relationships
- The syntax is as we might expect:

```
SELECT
e.name as employee_name,
m.name as manager_name

FROM people e
LEFT JOIN people m ON e.manager_id = m.id
```

FROM Questions? JOIN Questions?

Advanced Techniques:

String Manipulation

CROSS JOIN

Self Joins

UNION & UNION ALL

INTERSECT & EXCEPT

UNION & UNION ALL

- `UNION` and `UNION ALL` combine the results of two or more queries vertically (i.e. row-wise)
- `UNION ALL` keeps duplicate values, whereas `UNION` removes them
  - The difference between the two is one of the most common interview questions!

- `UNION` and `UNION ALL` require both/all queries to have the same number of columns
  - You could `UNION` unrelated columns if you had a specific use-case for it
    - Column names will come from the first query
  - In situations where you don't have exactly the same columns, but still need to `UNION`, you can pass a `NULL` (or zero, or blank) column
    - Similarly, you can pass a string character to keep track of which data is associated to which query

```
SELECT number_of_chips, number_of_tacos, 0 AS number_of_burritos, 'lunch' AS meal
FROM lunch

UNION

SELECT NULL as number_of_chips, number_of_tacos, number_of_burritos, 'dinner' AS meal
FROM dinner
```

UNION & UNION ALL

- If we recall SQLite's lack of support for `FULL OUTER JOINS`, `UNION ALL` will allow us to emulate one:

```
SELECT s1.quantity, s1.costume, s2.quantity
FROM store1 s1
LEFT JOIN store2 s2 ON s1.costume = s2.costume

UNION ALL

SELECT s1.quantity, s2.costume, s2.quantity
FROM store2 s2
LEFT JOIN store1 s1 ON s2.costume = s1.costume

WHERE s1.quantity IS NULL
```

UNION & UNION ALL

(`UNION` & `UNION ALL` live coding)

Any

Questions?

Advanced Techniques:

String Manipulation

CROSS JOIN

Self Joins

UNION & UNION ALL

INTERSECT & EXCEPT

INTERSECT & EXCEPT

- Both `INTERSECT` and `EXCEPT` require both/all queries to have the same number of columns

INTERSECT

- `INTERSECT` returns data in common with both/all `SELECT` statements
- Values returned will be distinct
- What's the difference between `INTERSECT` and `INNER JOIN`?

EXCEPT

- `EXCEPT` returns the opposite of an `INTERSECT`
  - for whatever rows are returned by the first `SELECT` statement, `EXCEPT` will return rows that were *not* returned by the second `SELECT` statement
- The "direction" of `EXCEPT` matters a lot
  - `EXCEPT` is relative the the first `SELECT` statement, so changing which comes first will always change the results of the query

INTERSECT & EXCEPT

Let's consider an example:

For the following `product` table,

product	product_id
blue bike	1
tiger onesie	2
house plant	3
headphones	4

and `order` table,

order_id	product_id
1	1
2	1
3	1
4	4

`INTERSECT` will find all products with work orders

```
SELECT product_id FROM product
INTERSECT
SELECT product_id FROM orders
```

Resulting in product_id's 1 & 4

`EXCEPT` will find all products *without* work orders

```
SELECT product_id FROM product
EXCEPT
SELECT product_id FROM orders
```

Resulting in product_id's 2 & 3

OR all work orders *without* products

```
SELECT product_id FROM orders
EXCEPT
SELECT product_id FROM product
```

Resulting in nothing (because no orders have a product_id that is not found in the product table)

INTERSECT & EXCEPT

(`INTERSECT` & `EXCEPT` live coding)

Questions?

Assignment 2: Analytic or Machine Learning Model

Q1) Create an analytic or machine learning model using our farmersmarket.db as the source of data.

For either, start with a question you want to answer. These are examples to get you thinking:

- Have sales for a vendor increased/decreased over time?
- Do some products sell better on rainy/snowy days?
- Will a customer who has purchased from a vendor return to that vendor and purchase again in the next 30 days?

Q2) Produce some sort of graphic from your SQL dataset.

If you have no experience making graphics, keep it simple (e.g. count by a variable). SQLite can support this using the Plot window.

If you feel more adventurous, use a tool like ggplot (R), seaborn (python), etc. If you have access to Excel or Tableau or another BI platform, export a csv from SQL and build there.

Describe why you picked this graphic. What does it tell us about your dataset?

Q3) Reflect: How can knowing SQL help you in your day-to-day work/life? Highlight a couple of different ways. This doesn't have to be strictly professional -- feel free to think about SQL in your personal life, if applicable.

Assignment 2: Analytic or Machine Learning Model

If you choose to make an analytic model, imagine that I have just taken over the administration of the farmers market but don't know any SQL. What do I need to know to be a good administrator? Focus on the following:

- Clean your data: e.g. if you have nulls, they should probably be coalesced.
- Include at least two tables (I'm not sure how you'd manage with fewer anyways). No one wants to remember IDs, so you should replace them.
- Include a performance metric associated with various time periods (e.g. years, quarters, etc).
- Discuss: Is the data model sufficient? Do we need other tables? Could we collect other data that might be helpful? If so, how would we gather this, and how would we store it (i.e. describe new tables in a logical or physical model)?
- Describe: How does your analytic model help me run the market? What insights can I gather from it?

Assignment 2: Analytic or Machine Learning Model

If you choose to make an ML model, pick a simple algorithm. This assumes you have some comfort in R or python. There is no expectation that you produce a statistically significant or highly accurate model. Focus on the following:

- Make at least one "feature" in SQL that serves your question (e.g. CASE statement, windowed function, etc).
- Include at least two tables in the query (I'm not sure how you'd manage with fewer anyways). IDs don't make for good features, so you should replace them.
- Report the accuracy or significance of your model.
- Discuss: How would you monitor this data? Can you collect "ground truth" data for your model? If yes, how would it flow back into SQL, and where would we store it (i.e. describe new tables in a logical or physical model)?
- Describe: Is your model supervised? Unsupervised? What type of output variable are you trying to solve for? Why did you pick this algorithm and why was it appropriate for the problem you were trying to describe?

Lingering Questions?