P

Final Review: Integration of NFL Statistics With Data Build Tool

Minhyuk Kang
Parker Jensen

Presentation Agenda

01 Dataset Introduction

What data did we collect for our dataset?

02 Data Constraint Creation

How have we structured our data so that it can be queried properly using dbt modeling?

03 Data Enrichment Modeling

How have we successfully represented our data enrichment using dbt modeling?

Dataset Introduction

What is Our Dataset?

For our solution, we chose to model several pieces of data related to the NFL.

We used four datasets to create our data warehouse. Three of them are dataset pulled from Kaggle, a website owned by Google that hosts user submitted datasets.

The other is extracted from an official document released by the Baltimore Ravens PR team, tracking statistics for the 2024 season.

What Are Our Tables?

Our dataset consists of Nine Tables:

nfl_stadiums - Models football stadiums entities.

spreadspoke_scores - Models individual football games over a large period of time

team_conference - Models the conference and division of individual teams.

team_identification - Models the name, abbreviated name, and id of a team.

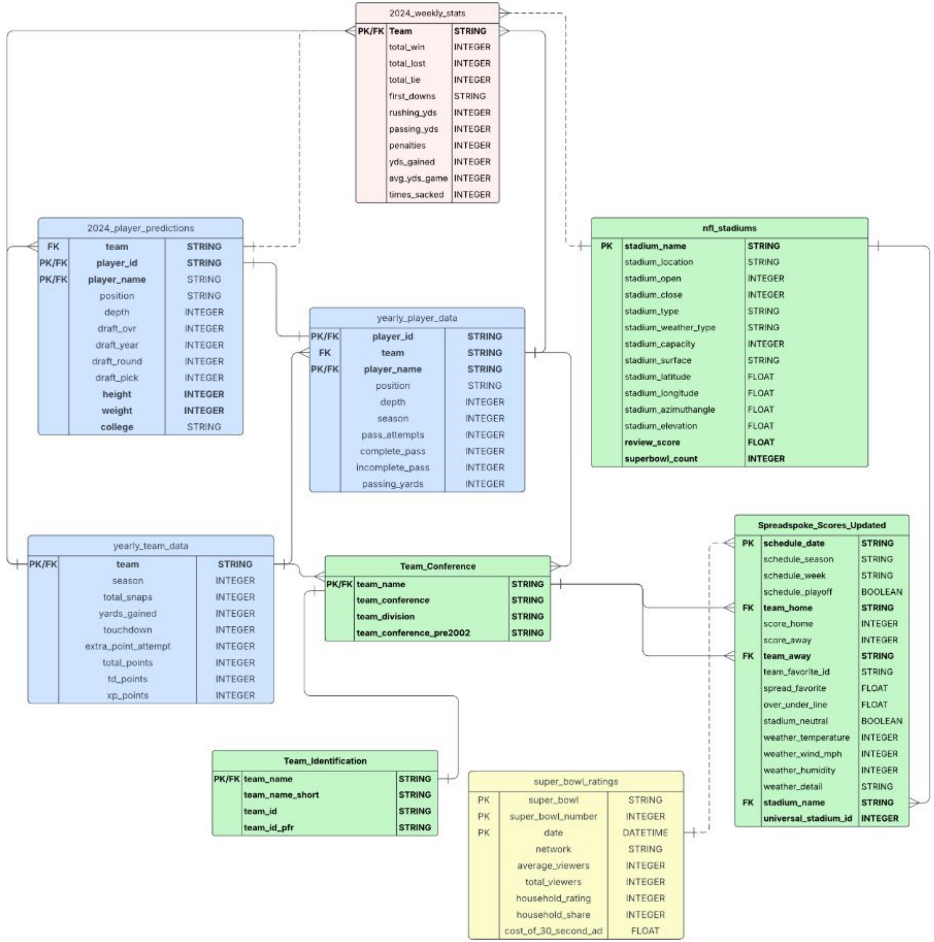
yearly_team_data - Models aggregate data for a team over a season, like total touchdowns, total points, etc.

yearly_player_data - Models aggregate data for a player over a season, like touchdowns, yards gained, etc.

2024_player_predictions - Models yearly predictions for player statistics on sports betting websites.

Super_bowl_ratings - Models individual super bowl games, focusing on the cost of advertisement, viewers, and television network hosting.

2024_weekly_stats - Models weekly stats for individual players in 2024. Extracted from a PDF file release by Baltimore Ravens PR.



Our entity relationship diagram (ERD) before dbt modeling. Different background shading indicates data from different sources.

Why Choose to Model This Data?

- NFL data has important implications for player careers, often influencing the decisions of team ownership in important contract negotiations.
- Furthermore, it’s helpful in fantasy football decision making, allowing one to construct a better average team
- NFL data has important connections to sports betting markets, where millions of dollars circulate and outcomes can be predicted and bet on based on average cases.
- Overall, NFL data allows us to make more informed decisions about what to do with players whether it be in betting markets, hobby fantasy football, or business decisions.

Introduction of Data Challenges

Interesting Challenges/Aspects of our Solution

The nature of our data led to unique constraints and schema requirements for DBT modeling.

Creating models that represent our data transformations led to unique lineage and several transformations per layer.

P

Data Constraint Creation

Problem: Our data needs to have primary and foreign key constraints in order to be useful.

Row	name	location	open	close	type	address	stadium_weather_type	stadium_cap...
1	Alamo Dome	San Antonio, TX	null	null	indoor	100 Montana St, San Antonio, T...	indoor	72000
2	Alamo Dome	San Antonio, TX	null	null	indoor	100 Montana St, San Antonio, T...	indoor	72000
3	Alamo Dome	San Antonio, TX	null	null	indoor	100 Montana St, San Antonio, T...	indoor	72000
4	Allianz Arena	Munich, Germany	null	null	outdoor	null	moderate	75024
5	Allianz Arena	Munich, Germany	null	null	outdoor	null	moderate	75024
6	Allianz Arena	Munich, Germany	null	null	outdoor	null	moderate	75024
7	Alltel Stadium	Jacksonville, FL	null	null	null	null	hot	null
8	Alltel Stadium	Jacksonville, FL	null	null	null	null	hot	null
9	Alltel Stadium	Jacksonville, FL	null	null	null	null	warm, hot	null
10	Alltel Stadium	Jacksonville, FL	null	null	null	null	hot	null
11	Alltel Stadium	Jacksonville, FL	null	null	null	null	warm, hot	null
12	Alltel Stadium	Jacksonville, FL	null	null	null	null	warm	null
13	Alltel Stadium	Jacksonville, FL	null	null	null	null	hot	null
14	Alumni Stadium	Chestnut Hill, MA	null	null	outdoor	Perimeter Rd, Chestnut Hill, MA ...	cold	null
15	Alumni Stadium	Chestnut Hill, MA	null	null	outdoor	Perimeter Rd, Chestnut Hill, MA ...	cold	null
16	Alumni Stadium	Chestnut Hill, MA	null	null	outdoor	Perimeter Rd, Chestnut Hill, MA ...	cold	null
17	Balboa Stadium	San Diego, CA	null	null	outdoor	Balboa Stadium, San Diego, CA ...	warm, hot	null
18	Balboa Stadium	San Diego, CA	null	null	outdoor	Balboa Stadium, San Diego, CA ...	hot	null
19	Balboa Stadium	San Diego, CA	null	null	outdoor	Balboa Stadium, San Diego, CA ...	warm	null
20	Balboa Stadium	San Diego, CA	null	null	outdoor	Balboa Stadium, San Diego, CA ...	hot	null
21	Balboa Stadium	San Diego, CA	null	null	outdoor	Balboa Stadium, San Diego, CA ...	warm, hot	null
22	Balboa Stadium	San Diego, CA	null	null	outdoor	Balboa Stadium, San Diego, CA ...	hot	null
23	Balboa Stadium	San Diego, CA	null	null	outdoor	Balboa Stadium, San Diego, CA ...	hot	null
24	Cotton Bowl	Dallas, TX	null	null	outdoor	1300 Robert B Cullum Blvd., Dal...	moderate	null
25	Cotton Bowl	Dallas, TX	null	null	outdoor	1300 Robert B Cullum Blvd., Dal...	moderate	null
26	Cotton Bowl	Dallas, TX	null	null	outdoor	1300 Robert B Cullum Blvd., Dal...	moderate	null
27	Dolphin Stadium	Miami, FL	null	null	null	null	warm, hot	null

A view of our table of Sports Stadiums. Many stadium names repeat themselves when they should not.

Problem: Our fields that are supposed to be foreign keys have many, many orphan entries because they often reference specific years games have taken place in.

Row	schedule_date	schedule_season	schedule_week	schedul...	team_home	score_home	score_away	team_away
1	12/11/1966	1966	14	false	Atlanta Falcons	16	10	St. Louis Cardinals
2	12/18/1966	1966	15	false	Atlanta Falcons	33	57	Pittsburgh Steelers
3	11/13/1966	1966	10	false	Atlanta Falcons	7	19	Baltimore Colts
4	9/11/1966	1966	1	false	Atlanta Falcons	14	19	Los Angeles Rams
5	10/30/1966	1966	8	false	Atlanta Falcons	17	49	Cleveland Browns
6	10/16/1966	1966	6	false	Atlanta Falcons	7	44	San Francisco 49ers
7	10/2/1966	1966	4	false	Atlanta Falcons	14	47	Dallas Cowboys
8	12/18/1966	1966	16	false	San Diego Chargers	17	27	Kansas City Chiefs
9	9/10/1966	1966	2	false	San Diego Chargers	24	0	New England Patriots
10	10/30/1966	1966	9	false	San Diego Chargers	24	17	Denver Broncos
11	10/2/1966	1966	5	false	San Diego Chargers	44	10	Miami Dolphins
12	9/4/1966	1966	1	false	San Diego Chargers	27	7	Buffalo Bills
13	12/11/1966	1966	15	false	San Diego Chargers	42	27	New York Jets
14	11/13/1966	1966	11	false	San Diego Chargers	19	41	Oakland Raiders
15	10/16/1966	1966	6	false	St. Louis Cardinals	10	10	Dallas Cowboys
16	9/11/1966	1966	1	false	St. Louis Cardinals	16	13	Philadelphia Eagles
17	11/27/1966	1966	12	false	St. Louis Cardinals	6	3	Pittsburgh Steelers
18	10/31/1966	1966	8	false	St. Louis Cardinals	24	17	Chicago Bears
19	12/17/1966	1966	15	false	St. Louis Cardinals	10	38	Cleveland Browns
20	10/9/1966	1966	5	false	St. Louis Cardinals	24	19	New York Giants
21	9/18/1966	1966	2	false	St. Louis Cardinals	23	7	Washington Redskins
22	11/20/1966	1966	11	false	Cleveland Browns	14	3	Washington Redskins
23	10/8/1966	1966	5	false	Cleveland Browns	41	10	Pittsburgh Steelers
24	9/18/1966	1966	2	false	Cleveland Browns	20	21	Green Bay Packers
25	11/13/1966	1966	10	false	Cleveland Browns	27	7	Philadelphia Eagles
26	9/25/1966	1966	3	false	Cleveland Browns	28	34	St. Louis Cardinals
27	10/23/1966	1966	7	false	Cleveland Browns	30	21	Dallas Cowboys

A view of our table of spreadspoke scores. Notice that games take place in many different years, but we only have data in other tables from 2024.

So How Do We Create These Constraints?

- There are several different cases where we need to create constraints, so different approaches should be used to ensure our data is kept intact.
- For example, we have situations where our primary key fields should repeat, situations where it shouldn't.
- For foreign keys, we have situations where we want to remove orphan records and situations where we don't.

Altering our datasets to create these constraints is quite simple, and can be done easily through dbt modelling. What's important is choosing how exactly to alter our data.

Methodology For Constraint Creation

- For primary keys we want to be unique, we will select each unique value and reduce the table in size to only unique rows (this works because our dataset does not have rows with the same foreign key but different data inside them).
- For primary keys we want to repeat, we'll instead create a new identifier to act as primary key for querying.
- For foreign keys where we don't want orphan records, we'll simply delete the orphan records.
- For foreign keys that have many, many orphan records, we simply won't use a foreign key constraint as it doesn't properly apply in this case.

Example Constraint Creation

- Schema and constraints in dbt are stored in a .yml format.
- As shown here, we define all our tables and their data types, and we add any constraints using the appropriate flag.
- We also add the appropriate tests to make sure we are properly testing if our constraints are applied correctly.

```yaml
version: 2

models:
  - name: team_conference
    config:
      contract:
        enforced: true
    columns:
      - name: name
        data_type: string
        constraints:
          - type: primary_key
          - type: foreign_key
            to: ref('team_identification')
            to_columns: [name]
        tests:
          - unique
          - not_null
          - relationships:
              to: ref('team_identification')
              field: name
      - name: conference
        data_type: string
      - name: conference_pre_2002
        data_type: string
      - name: division_pre_2002
        data_type: string
      - name: _data_source
        data_type: string
      - name: _load_time
        data_type: timestamp
```

Implementation For Unique Primary Key

- First, we create a new dbt sql model to create our table.
- Then, we simply define a temporary table as all distinct rows from the raw data.
- Note that we exclude load time here, as it is always unique per row and would give us repeat primary keys. We will add it back into the dataset immediately after in a python model.

```
with stg_superbowl_ratings as(
    select distinct * except(_load_time)
    from {{ source ('football_dataset_raw', 'superbowl_ratings')}}
)

select *
from stg_superbowl_ratings
```

```
from pyspark.sql.functions import current_timestamp, lit

def model(dbt, session):
    stats_df = dbt.ref('stg_tmp_superbowl_ratings')
    stats_df = stats_df.withColumn("_load_time", lit(current_timestamp()))
    return stats_df
```

Implementation For Repeated Primary Key

- First, we create a new dbt sql model to create our table.
- Then, we simply define a temporary table as all distinct rows from the raw data, excluding load time like before.
- Now, in the python model, in addition to adding load time back, we add a new field to the table simply called “game_id” that is unique to every row, allowing for primary key constraint.

```
with stg_spreadspoke_scores as(
    select distinct * except (_load_time)
    from {{ source ('football_dataset_raw', 'spreadspoke_scores')}}
)

select *
from stg_spreadspoke_scores
```

```
from pyspark.sql.functions import current_timestamp, lit, monotonically_increasing_id

def model(dbt, session):
    stats_df = dbt.ref('stg_tmp_spreadspoke_scores')
    stats_df = stats_df.withColumn("_load_time", lit(current_timestamp()))
    stats_df = stats_df.withColumn("game_id", monotonically_increasing_id())
    return stats_df
```

Implementation For Removing Orphan Keys

- First, define a post hook, a sql function that will execute after the model is created and the table is defined. This post hook simply deletes all entries with foreign keys that aren't found in the table they are supposed to reference.
- Then, we create the sql table in this sql model by selecting everything from the staging layer table.

```
{{ config(
    post_hook = "delete from {{ this }} where team not in (select name from {{ref('team_conference')}})"
)}}

with int_2024_weekly_stats as(
    select * from {{ref('2024_weekly_stats')}}
)

select *
from int_2024_weekly_stats
```

Implementation For Not Removing Orphan Keys

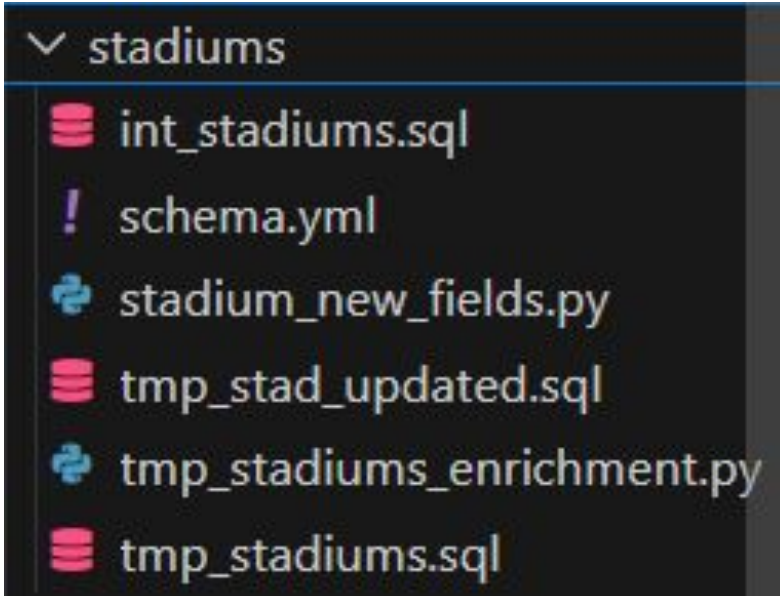
- The solution to this is extremely simple, we just don’t add a foreign key constraint to this specific field, but why?.
- Our spreadspoke scores table references over 60 years of football games, while our other tables only reference one, 2024.\
- On a theoretical level the dates in these tables shouldn’t have a foreign key relationship.
- On a practical level, deleting orphan entries would mean deleting 99% of the spreadspoke scores table, over 14000 entries, and we want to preserve this data for analysis.
- Notice the lack of foreign key constraint on the schedule_date field as shown here.

```yaml
models:
  - name: int_spreadspoke_scores
    config:
      contract:
        enforced: true
    columns:
      - name: schedule_date
        data_type: string
      - name: schedule_season
        data_type: string
      - name: schedule_week
        data_type: string
      - name: schedule_playoff
        data_type: boolean
      - name: team_home
        data_type: string
        constraints:
          - type: foreign_key
            to: ref('team_conference')
            to_columns: [name]
        tests:
          - relationships:
              to: ref('team_conference')
              field: name
```

Code Execution

Data Enrichment Modeling

Problem: Our Stadium Table requires multiple transformations and steps of enrichment.



A view of our folder of stadium models for the intermediate layer. Notice how many models there are.

How Do We Model These Transformations Properly?

- This solution is not very difficult or challenging, but it is interesting seeing how all of these come together and how we model these sorts of successive transformations in dbt with things like the ref function.
- We have 4 transformations that need to occur, then one final model to represent the final intermediate table.

The solution to this is very simple, but let's take a look at how we ended up at our final stadiums table.

Implementation

- First, create a temporary table of just stadium names and give it to a python model that will enrich our data through the use of an LLM.
- This was the subject of our last presentation, so it won't be gone over in detail, just know that after the python model is done, we have a table named tmp_stadiums_enrichment that contains unique rows of stadium names, locations, addresses, etc. This is to fill the holes in our stadium table with data that should be there.

```python
def model(dbt, session):
    input_df = dbt.ref("tmp_stadiums")
    num_stadiums = input_df.count()
    print("number of stadiums to process: ", num_stadiums)

    batch_size = 30
    num_batches = int(num_stadiums / batch_size)
    combined_results = []

    pandas_df = input_df.select("name", "location").filter("name is not null and location is not null").toPandas()
    batches = numpy.array_split(pandas_df, num_batches)

    for i in range(num_batches):
        subset_stadiums = batches[i].to_string(header = False)
        results = enrich(subset_stadiums)
        combined_results.extend(results)

    schema = StructType([
        StructField("name", StringType(), True),
        StructField("location", StringType(), True),
        StructField("address", StringType(), True),
        StructField("opening_date", StringType(), True),
        StructField("roof_type", StringType(), True),
        StructField("capacity", IntegerType(), True),
        StructField("longitude", FloatType(), True),
        StructField("latitude", FloatType(), True)
        ])

    output_df = session.createDataFrame(combined_results, schema)
    output_df = output_df.withColumn("_data_source", lit("Kaggle"))
    output_df = output_df.withColumn("_load_time", lit(current_timestamp()))
    num_stadiums = output_df.count()

    print("number of stadiums returned:  ", num_stadiums)

    return output_df
```

```python
import itertools, json, pandas
from jsonschema import validate
import numpy
from pyspark.sql.types import StructField, StructType, IntegerType, StringType, FloatType
from pyspark.sql.functions import current_timestamp, lit
import vertexai
from vertexai.generative_models import GenerativeModel, Part

region = "us-central1"
model_name = "gemini-2.0-flash-001"
prompt = """Here is a list of names.
I want you to check of the name corresponds to an American Football Stadium.
if it does, return the name of the stadium, the state and city it is from, its full address, the year it opened,,
whether it has an open, closed, or retractable roof, its capacity, its longitude, and its latitude.
Return the reuslts as a list of JSON objects with the schema [{"name" : string, "location" : string, "address" : st
Return only one answer per stadium.
Do include stadiums that have already been closed.
Do not include commas in the capacity field.
Return the opening_date field as a string.
Return Longitude and Latitude as a signed floating point number.
the roof_type field should be entirely lowercase.
Don't return the records which are not American football stadiums.
Don't return any empty JSON objects.
Don't return an explanation for your answer.

Here are some sample runs:

I give you:
"Fenway Park, Boston, MA"
"SoFi Stadium, Inglewood, CA"
"Allegiant Stadium, Paradise, NV"

You return:
{"name": "SoFi Stadium", "location": "Inglewood, CA", "address": "1001 S. Stadium Dr. Inglewood, CA 90301", "openin
{"name": "Allegiant Stadium", "location": "Las Vegas, NV","address": "3333 Al Davis Way Las Vegas, NV 89118", "oper
"""
```

Implementation

- Next, we create a new stadium mapping table to add the universal stadium id from our previous project work to the stadium table (as well as others for foreign key constraint).
- This results in a table of stadium mappings by stadium name we can use in other implementation.

```
with stadiums as (
    select 'name' from {{ ref('tmp_stadiums_enrichment') }}
),
spreadspoke as (
    select stadium from {{ ref('spreadspoke_scores') }}
),
all_names as (
    select 'name' as original_name from {{ ref('stadiums') }}
    union all
    select stadium as original_name from {{ ref('spreadspoke_scores') }}
),
deduped_names as (
    select distinct original_name from all_names
),
numbered_names as (
    select
        original_name,
        row_number() over (order by original_name) as universal_stadium_id
    from deduped_names
)

select * from numbered_names
```

Implementation

- Next, we create a new stadium updated table that joins the mapping and enrichment tables together, so we can have all of their data in one table.
- We now have a table of unique stadium rows with their universal stadium IDs, altogether for future model creation.

```sql
with stadiums as (
    select * from {{ ref('tmp_stadiums_enrichment') }}
),
mapping as (
    select * from {{ ref('tmp_stad_mapping') }}
),
joined as (
    select
        s.* except (name),
        m.universal_stadium_id,
        m.original_name as stadium_name
    from stadiums s
    left join mapping m
    on s.name = m.original_name
)

select * from joined
```

Implementation

- Like last time, we also are adding new fields to the stadium table of stadium review score and superbowl count.
- We take our earlier temporary table of just stadium names and give it to an LLM for enrichment. Much like the other LLM transformation, this was gone over last presentation so it won't be in detail.
- Know that after this model is done, we'll have a sql table of unique names, review scores, and superbowls hosted.

```python
import itertools, json, pandas
from jsonschema import validate
import numpy
from pyspark.sql.types import StructField, StructType, IntegerType, StringType, FloatType
from pyspark.sql.functions import current_timestamp, lit
import vertexai
from vertexai.generative_models import GenerativeModel, Part

region = "us-central1"
model_name = "gemini-2.0-flash-001"
prompt = """Here is a list of names of American football stadiums.
  I want you to take the name of the stadium and return each stadium's name, its average review score on Google, as
  Return the results as a list of properly formatted JSON objects.
  Return only one answer per stadium.
  You must return the public review score of each stadium found on Google.
  You must return how many superbowl games each stadium has hosted.
  Format each element in the list as a JSON object with the schema:
  {"name": string, "review_score": float, "superbowls_hosted": integer}
  return the name as a string.
  return the review score as a float.
  round the review score to one decimal point.
  return the superbowl count as an integer.
  do not return any null values for review_score.
  do not return any null values for superbowl_count.
  if you cannot find the review score of a stadium, return 0.
  do not include an explanation with your answer.
  """
def model(dbt, session):
    input_df = dbt.ref("tmp_stadiums_enrichment")
    num_stadiums = input_df.count()
    print("number of stadiums to process: ", num_stadiums)

    batch_size = 30
    num_batches = int(num_stadiums / batch_size)
    combined_results = []
    pandas_df = input_df.select("name").filter("name is not null").toPandas()
    batches = numpy.array_split(pandas_df, num_batches)

    for i in range(num_batches):
        subset_stadiums = batches[i].to_string(header = False)
        results = enrich(subset_stadiums)
        combined_results.extend(results)

    schema = StructType([
        StructField("name", StringType(), True),
        StructField("review_score", FloatType(), True),
        StructField("superbowls_hosted", IntegerType(), True)
    ])

    output_df = session.createDataFrame(combined_results, schema)
    num_stadiums = output_df.count()

    print("number of stadiums returned:  ", num_stadiums)

    return output_df
```

Implementation

- Lastly, we join the new fields table to the updated stadiums table so we can finally have all the data together.
- We also create a new schema.yml file to ensure proper schema formatting for this final intermediate layer table
- We now have a stadiums table with unique rows, filled in with the proper data, and enriched with mapping IDs and new fields of information to query for and analyze.

```
with int_stadiums as (
    select distinct updated.*,
    new_fields.* except(name)
    from {{ref('stadium_new_fields')}} new_fields
    join {{ref('tmp_stad_updated')}} updated
    on new_fields.name = updated.stadium_name
)

select *
from int_stadiums
```

Thank you!