CITS5504 Project1 Report - Sally Chen (23687599)

1 Introduction

In this study, I delve into the question of the economic impact of the Olympic Games on different countries and the link between demographics and sporting activity. To support this complex data analysis requirement, I designed a series of fact sheets and dimension tables and carried out careful cleaning and selective rounding of data sources.

Particularly, I concentrate on post-1980, reflecting on the Olympics' role in the contemporary development of countries and cities. This temporal focus necessitates the exclusion of data preceding the year 1980 from my analysis.

1.1 Rationale for data source cleaning and selective rounding

Economic data: Containing key economic indicators such as the country's GDP per capita, etc., By filtering and collating this data, it is possible to directly correlate economic conditions with Olympic performance in order to answer complex questions such as the relationship between government health spending and Olympic performance.

Global Population: By analysing the total population, it is possible to explore the relationship between population size and the number of Olympic medals, and to explore how populous countries can promote sport to become major sporting nations.

Olympic data: contains medal data and information on the staging of the Olympic Games, and is an indispensable data source for analysing the impact of the Games on each participating country. The detailed records of medal data enable in-depth analyses of the performance of different countries in the Olympic Games and its relationship with economic and demographic factors.

1.2 Compositional considerations for fact and dimension tables

In order to effectively support this analysis, the following fact tables and dimension tables have been designed:

Olympic Fact Table records Olympic detailed performance of individual athletes and provide micro data to support the analysis. **2020 Economic Data Fact Table** focuses on recording national economic indicators, such as GDP per capita , health spending, population, and life expectancy, which are critical to understanding the economic impact of the Olympics. **Location Dimension Table, Year Dimension Table**, **OlympicGames Dimension Table**, **Event Dimension Table**, and **Player Dimension Table** provide detailed contextual information about location, time, and details of different countries' performance in Olympic Games , allowing data analysis to be performed on multiple dimensions.

1.3 Clients Chosen

1.3.1 Client A: Sports Economics Researcher

Client A is dedicated to the link between the Olympic Games and the economy. Including the relationship between the number of medals and GDP per capita in different countries. The correlation between economic data and Olympic performance in different countries was analysed.

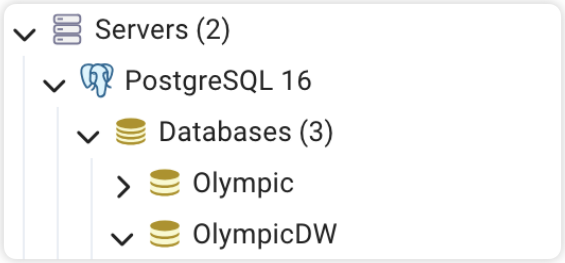
1.3.2 Client B: Sports Promotion Officer

Client B is concerned with understanding the population of different countries and the link between population, geography and sporting activity. By examining the relationship between population size, life expectancy and the number of Olympic medals, Client B explored the role of major sporting events in promoting the popularity of sporting activities.

2 OLTP Database Loading

2.1 Extract Data

I personally built a server that is going to store data from data source and data warehouse. The content of this project is done locally.




```
# use pandas to connect and read SQL
import pandas as pd
from sqlalchemy import create_engine

connection_url = f"postgresql://postgres:....@localhost:1225/Olympic"
engine = create_engine(connection_url)

import chardet

# read the csv files and check the encoding and decode
csv_files = ['Olympic/Economic data.csv','Olympic/Global Population.csv','Olympic/life-expectancy.csv','Olympic/list-of-countries_areas-by-continent-2024.csv','Olympic/mental-illness.csv','Olympic/olympic_hosts.csv','Olympic/olympic_medals.csv']

# read the csv and decode
for file in csv_files:
    with open(file, 'rb') as f:
        result = chardet.detect(f.read())
    encoding = result['encoding']
    df = pd.read_csv(file, encoding=encoding)
    table_name = file.split('/')[-1].replace('.csv', '')
    df.to_sql(table_name, engine, if_exists='replace', index=False)
```

2.2 Transform

2.2.1 Table 1: olympic_medals

```
# check 1/6 table - olympic_medals
medals_data = pd.read_sql_table('olympic_medals', engine)
medals_data.to_sql('olympic_medals_backup', engine, if_exists='replace', index=False)
print(medals_data.head())

# delete 'participant_title' , 'athlete_url' and 'country_code'
```

```
medals_data.drop(['participant_title', 'athlete_url', 'country_code'], axis=1,
inplace=True)

# clean the data of gameteam: de-duplication of medals
team_medals = medals_data[medals_data['participant_type'] == 'GameTeam']
team_medals_unique = team_medals.drop_duplicates(subset=['country_name', 'medal_type',
'discipline_title', 'event_title'])
medals_data = medals_data[medals_data['participant_type'] != 'GameTeam']
medals_data_cleaned = pd.concat([medals_data, team_medals_unique], ignore_index=True)

# check null value
print(medals_data_cleaned.isnull().sum())

# Use value of 'country_3_letter_code' to fullfill null data of 'athlete_full_name'
medals_data_cleaned['athlete_full_name'].fillna(medals_data_cleaned['country_3_letter_code
'], inplace=True)

# check the types of data
print(medals_data_cleaned.info())

# set data type to category
medals_data_cleaned['medal_type'] = medals_data_cleaned['medal_type'].astype('category')
medals_data_cleaned['event_gender'] =
medals_data_cleaned['event_gender'].astype('category')
medals_data_cleaned['participant_type'] =
medals_data_cleaned['participant_type'].astype('category')
medals_data_cleaned['country_3_letter_code'] =
medals_data_cleaned['country_3_letter_code'].astype(str)

# rename the columns for OLAP
medals_data_cleaned.rename(columns={
    'discipline_title': 'DisciplineTitle',
    'slug_game': 'OlympicGame',
    'event_title': 'EventTitle',
    'event_gender': 'AthleteGender',
    'medal_type': 'MedalType',
    'participant_type': 'ParticipantType',
    'athlete_full_name': 'AthleteName',
    'country_name': 'CountryName',
    'country_3_letter_code': 'CountryCode'
}, inplace=True)

# write into db
medals_data_cleaned.to_sql('olympic_medals', engine, if_exists='replace', index=False)
print(medals_data_cleaned.describe())
```

> I did a backup of the source data at thte very beginning. I removed the `participant_title`, `athlete_url`, and `country_code` from the original `olympic_medals` dataset, which were not necessary for the purpose of my analysis, and their removal simplified the structure of the dataset and reduced potential confounds. Specifically, I didn't remove medal records prior to 1980 by hand, this is because the data in the dataset doesn't have a year column. Will do that later when forming fact

> and dimension tables.
>
> When dealing with the medal data for the team events, I paid particular attention to the issue of medal de-weighting. Since medals in team events are counted only once in the rankings, direct counting would lead to double counting and distortion of the data. Therefore, I de-duplicated the medals for team events to ensure that medals for each team event were counted only once.
>
> In addition, I filled in the blank values for the team event athlete names, using `country_3_letter_code` as a replacement, which preserves the nationality information of the participants and avoids the problem of null values in the data. I also adjusted the types of the data fields, setting `medal_type`, `event_gender`, and `participant_type` to category types. Finally, I renamed the column names of the dataset to better match the needs of the OLAP operation.

2.2.2 Table 2: olympic_hosts

```
# check 2/6 table - olympic_hosts
hosts_data = pd.read_sql_table('olympic_hosts', engine)
hosts_data.to_sql('olympic_hosts_backup', engine, if_exists='replace', index=False)

# check null value
print(hosts_data.isnull().sum())

# delete 'game_end_date' and 'game_start_date'
hosts_data.drop(['game_end_date', 'game_start_date'], axis=1, inplace=True)

# delete Olympic Games before 1980
hosts_data = hosts_data[:23]

# add a new column and name it 'HostCityName'
hosts_data['HostCityName'] = hosts_data['game_name'].apply(lambda x: " ".join(x.split()
[:-1]))

# delete 'game_name'
hosts_data.drop(['game_name'], axis=1, inplace=True)

# check the types of data
print(hosts_data.info())

# set data type
hosts_data['game_season'] = hosts_data['game_season'].astype('category')
hosts_data['game_year'] = hosts_data['game_year'].astype(int)

# rename the columns for OLAP
hosts_data.rename(columns={
    'game_slug': 'OlympicGame',
    'game_location': 'CountryName',
    'game_season': 'OlympicSeason',
    'game_year': 'Year',
}, inplace=True)

# write into db
```

```
hosts_data.to_sql('olympic_hosts', engine, if_exists='replace', index=False)
```

> In the process of cleaning the `olympic_host` data table, I removed Olympic Games records prior to 1980 because the focus of the research is to explore how the Olympic Games have influenced the modernisation of countries and cities after 1980. A new column `HostCityName` was added by extracting the host city name from the `game_name field`.

2.2.3 Table 3: list-of-countries_areas-by-continent-2024

```
# check 3/6 table - list-of-countries_areas-by-continent-2024
countries_data = pd.read_sql_table('list-of-countries_areas-by-continent-2024', engine)
countries_data.to_sql('list-of-countries_areas-by-continent-2024_backup', engine,
if_exists='replace', index=False)

# check null value
print(countries_data.isnull().sum())

# check the types of data
print(countries_data.info())

# set data type
countries_data['region'] = countries_data['region'].astype('category')

# rename the columns for OLAP
countries_data.rename(columns={
    'country': 'CountryName',
    'region': 'ContinentName'
}, inplace=True)

# write into db
countries_data.to_sql('list-of-countries_areas-by-continent-2024', engine,
if_exists='replace', index=False)
```

> For special regions such as Taiwan and Hong Kong, which use different entry names in the Olympics, such as `Chinese Taipei` and `Hong Kong, China`, these differences require special attention in subsequent data cleansing. In order to ensure that our data warehouse correctly reflects the official records and statistics of the Olympic Games, these name inconsistencies will be appropriately handled during the creation of dimension tables to ensure data accuracy and consistency.

2.2.4 Table 4: Economic data

```
# check 4/6 table - Economic data
economic_data = pd.read_sql_table('Economic data', engine)
economic_data.to_sql('Economic data_backup', engine, if_exists='replace', index=False)

# check null value
print(economic_data.isnull().sum())

print(economic_data.head())
```

```
# delete the last five lines
economic_data = economic_data[:-5]

# delete unnecessary coumns
economic_data.drop(['Time Code', 'Poverty headcount ratio at $2.15 a day (2017 PPP) (% of
populat', 'GDP per capita growth (annual %) [NY.GDP.PCAP.KD.ZG]', 'Secure Internet servers
(per 1 million people) [IT.NET.SECR.P6]', 'Mortality rate, infant (per 1,000 live births)
[SP.DYN.IMRT.IN]', 'Domestic general government health expenditure per capita (curr',
'Domestic private health expenditure per capita (current US$) [S', 'External health
expenditure per capita (current US$) [SH.XPD.EH'], axis=1, inplace=True)

# replace ".." into NaN
economic_data.replace('..', pd.NA, inplace=True)

# check the types of data
print(economic_data.info())

# rename the columns for OLAP
economic_data.rename(columns={
    'Time': 'Year',
    'Country Name': 'CountryName',
    'Country Code': 'CountryCode',
    'GDP per capita (current US$) [NY.GDP.PCAP.CD]': 'GDP',
    'Current health expenditure (% of GDP) [SH.XPD.CHEX.GD.ZS]': 'HealthGDP'
}, inplace=True)
print(economic_data.head())

# set data type to float and round to 2 decimal
economic_data['GDP'] = pd.to_numeric(economic_data['GDP']).round(2)
economic_data['HealthGDP'] = pd.to_numeric(economic_data['HealthGDP']).round(2)

economic_data['Year'] = economic_data['Year'].astype(int)

# write into db
economic_data.to_sql('Economic data', engine, if_exists='replace', index=False)
```

> After reviewing the data source, I decided to remove the last five rows and a number of unnecessary columns as they did not fit my analytical objectives.
>
> When working with the data types, I paid particular attention to the raw data with `..` to denote missing values. To accurately convert these missing values to float types, I used the `pd.to_numeric` function. Additionally, I rounded the economic data to two decimal places to ensure consistency and readability.

2.2.5 Table 5: Global Population

```
# check 5/6 table - Global Population
population_data = pd.read_sql_table('Global Population', engine)
population_data.to_sql('Global Population_backup', engine, if_exists='replace',
index=False)
```

```
# check null value
print(population_data.isnull().sum())

# delete unnecessary lines and columns, only want data from 2020
population_data = population_data.drop(0)
population_data = population_data[:-34]
population_data = population_data[['Population (Millions of people)', '2020']]

# replace "no data" into NaN
population_data = population_data.replace('no data', pd.NA)

# check the types of data
print(population_data.info())

# rename the columns for OLAP
population_data.rename(columns={
    'Population (Millions of people)': 'CountryName'
}, inplace=True)

# set data type to float and round to 2 decimal
population_data['2020'] = pd.to_numeric(population_data['2020']).round(2)

# write into db
population_data.to_sql('Global Population', engine, if_exists='replace', index=False)
```

> I only kept data of 2020. Because I would merge it into Economic Data later and Economic Data only have data from 2020.
>
> The original data contained population data broken down by continent, which led to data duplication and categorisation complications, such as the presence of both 'Asia' and 'East Asia'. To simplify the data model, I chose to remove this data.

2.2.6 Table 6: life-expectancy

```
# check 6/6 table - life-expectancy
life_expectancy_data = pd.read_sql_table('life-expectancy', engine)
life_expectancy_data.to_sql('life-expectancy_backup', engine, if_exists='replace',
index=False)

# check null value
print(life_expectancy_data.isnull().sum())

# delete life expectancy of regions that don't have a country code
life_expectancy_data = life_expectancy_data.dropna(subset=['Code'])

# keep life expectancy of year 2020
life_expectancy_data = life_expectancy_data[life_expectancy_data['Year'] == 2020]

# rename the columns for OLAP
life_expectancy_data.rename(columns={
    'Entity': 'CountryName',
```

```
    'Code': 'CountryCode',
    'Period life expectancy at birth - Sex: all - Age: 0': 'LifeExpectancy'
}, inplace=True)

# set data type to float and round to 2 decimal
life_expectancy_data['LifeExpectancy'] =
pd.to_numeric(life_expectancy_data['LifeExpectancy']).round(2)

# write into db
life_expectancy_data.to_sql('life-expectancy', engine, if_exists='replace', index=False)
```

> I only kept data of 2020. Because I would merge it into Economic Data later and Economic Data only have data from 2020.

3 OLAP Database Loading

3.1 Warehouse Design

Defining conceptual hierarchies in a dimension table can support queries at different levels of granularity. For example, in the `Location Dimension`, there may be a `country-continent` hierarchy that allows queries to start at the finest level of granularity (country) and then summarise the data by continent.
The `Year Dimension` hierarchy supports queries based on a specific year or on whether it is an Olympic year.
The `Olympic Games Dimension` hierarchy supports queries between Olympic seasons (winter or summer), specific Olympic Games names, and host cities.
The `Event Dimension` hierarchy allows data to be viewed between specific sport disciplines and sub-events, with the ability to select the granularity of the query as required, e.g. individual events, or to aggregate the data to a higher level, e.g. by sport or Olympic season.

Due to the specificity of the DATA SOURCE, `Economic Fact Data` is only available for the year `2020`, which somewhat limits the breakdown of the hierarchy as well as the aggregated queries. Despite the existence of `Year Dimension`, the main research still focuses on the economic performance in 2020 and the performance of the Olympic Games, which makes it difficult to study the impact of the Olympic Games on the economic development of a specific country in more depth. For example, the change in GDP before and after hosting the Olympics.

3.1.1 Fact Tables

3.1.1.1 Olympic Fact Table

- **RecordID** (Primary Key): Unique identifier
- **CountryCode** (Foreign Key): Country code
- **YearID** (Foreign Key): Olympic year
- **OlympicGame** (Foreign Key): Name of the Olympic Games
- **EventID** (Foreign Key): Unique identifier for the event: name of a event title
- **AthleteID**: (Foreign Key): Unique identifier for the athlete
- **MedalType**: Gold, Silver, Bronze

3.1.1.2 2020 Economic Data Fact Table

- **EconomicFactID** (Primary Key): Unique identifier
- **CountryCode** (Foreign Key): Country code
- **YearID** (Foreign Key): Olympic year
- **GDP**: GDP per capita
- **HealthGDP**: Health expenditure GDP
- **Population**: Population in 2020
- **LifeExpectancy**: Life expectancy of 2020

3.1.2 Dimension Tables

3.1.2.1 Location Dimension Table

- **CountryCode** (Primary Key): Country code
- **CountryName**: Name of the country
- **ContinentName**: Continent (e.g., Asia, Europe)

3.1.2.2 Year Dimension Table

- **YearID** (Primary Key): Year
- **Year** : Year for analysis
- **OlympicYearIndicator**: Indicates whether it is an Olympic year (yes/no)

3.1.2.3 Olympic Games Dimension Table

- **OlympicGame** (Primary Key): Unique identifier for the event: name of the Olympic Games
- **CountryCode**: Country code
- **HostCityName**: Name of the host city
- **OlympicSeason**: Type of Olympic Games (Summer, Winter)

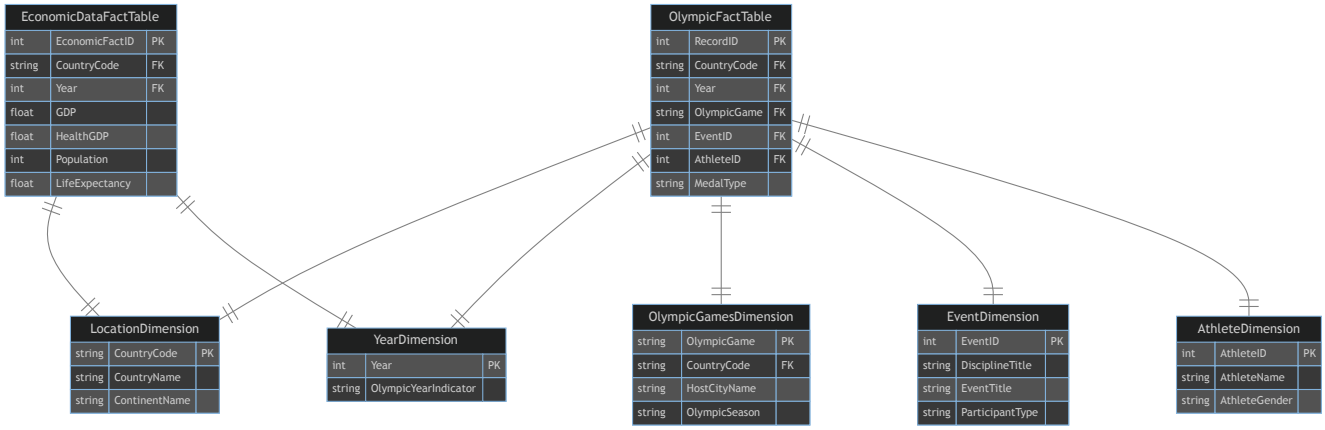
3.1.2.4 Event Dimention Table

- **EventID** (Primary Key): Unique identifier for the event: name of a event title
- **DisciplineTitle**: Name of discipline title
- **EventTitle**: Name of event title
- **ParticipantType** : Athlete or GameTeam

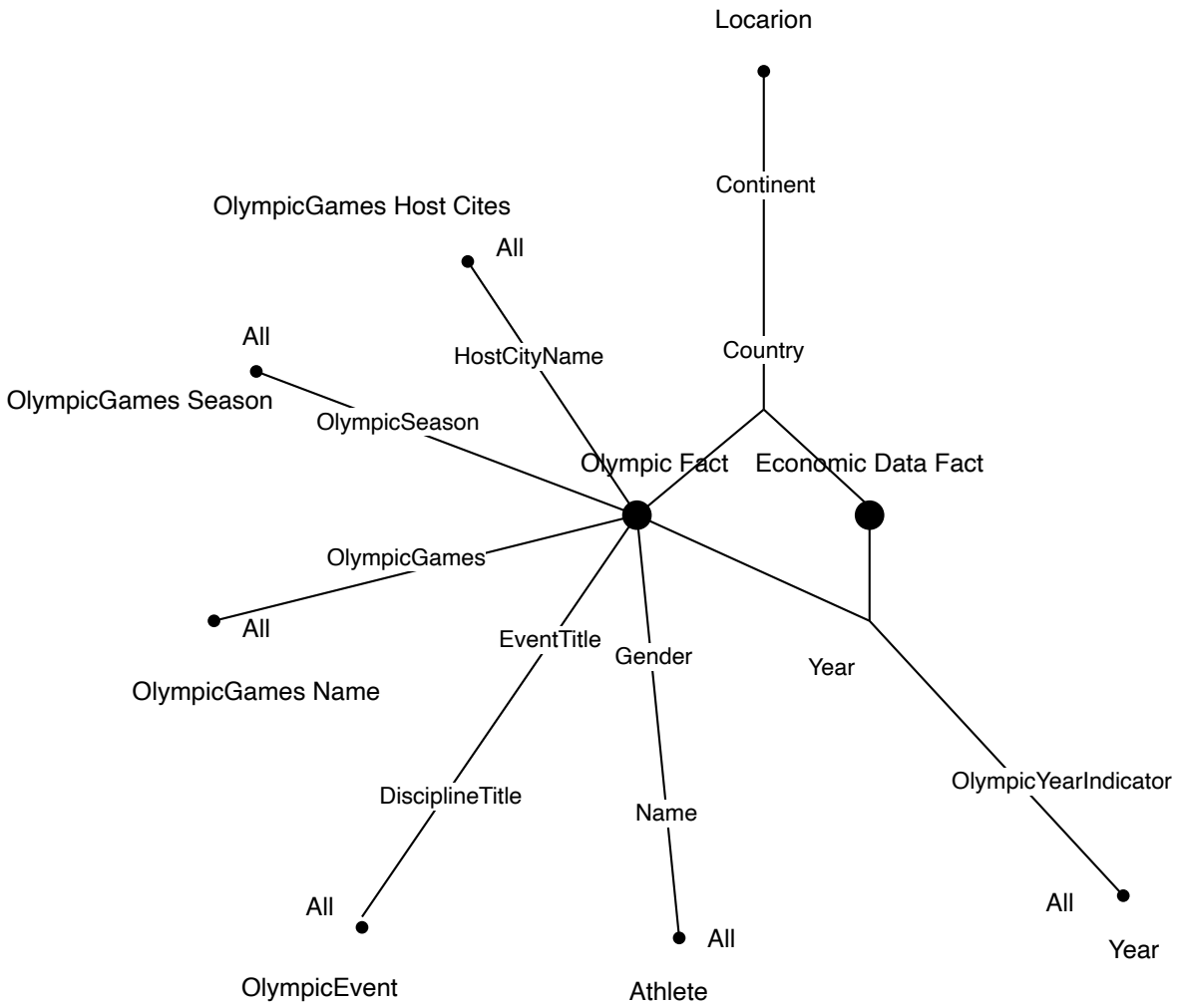
3.1.2.5 Athlete Dimension Table

- **AthleteID**: (Primary Key): Unique identifier for the athlete
- **AthleteName**: Name for an athlete or country name for a gameteam
- **AthleteGender**： Gender for an athlete or Mixed gender for a gameteam

3.2 ER - Star Schema



3.3 Starnet Query Model



3.4 ETL

My data cleaning process is in support of the study of the performance of the Olympic Games in different countries and economic data for the year 2020. So the first step was to merge and clean the DimLocation tables. This step is foundational because countries that do not exist in either fact table will be excluded from my analysis. Therefore, the list of countries is the basis of my warehouse.

Subsequently, ensuring the consistency and uniformity of the country names in the tables is the most critical step in the process. I used country names from Wikipedia and applied manual corrections to ensure accuracy. This standardisation ensured consistency throughout my data warehouse.

3.4.1 Location Dimension Table

```
# Location Dimension Table
# need CountryCode, CountryName from Economic data
DimLocation = pd.read_sql_table('Economic data', engine)

# copy into DW
DimLocation.to_sql('DimLocation', new_engine, if_exists='replace', index=False)

# delete unnecessary columns
DimLocation.drop(['Year', 'GDP', 'HealthGDP'], axis=1, inplace=True)

# confirm CountryName
# handle consistencies for special regions such as Hong Kong in DimLocation
DimOlympicGames = pd.read_sql_table('olympic_medals', engine)

# create a DataFrame of unique country codes and names from the olympic_medals table
unique_countries = DimOlympicGames[['CountryCode', 'CountryName']].drop_duplicates()

# merge to find inconsistencies
DimLocation = pd.merge(DimLocation, unique_countries, on='CountryCode', how='left',
suffixes=('', '_olympic'))

# find inconsistencies
inconsistencies = DimLocation[DimLocation['CountryName'] !=
DimLocation['CountryName_olympic']]

# print inconsistencies
print(inconsistencies[['CountryCode', 'CountryName', 'CountryName_olympic']])

# delete rows with NA in CountryName_olympic
DimLocation = DimLocation[~DimLocation['CountryName_olympic'].isna()]

# find rows where CountryName and CountryName_olympic are inconsistent
inconsistencies = DimLocation[DimLocation['CountryName'] !=
DimLocation['CountryName_olympic']]

# print and check
print(inconsistencies[['CountryCode', 'CountryName', 'CountryName_olympic']])

# set up a dictionary with correct country code and name
corrections = {
    'BRN': {'new_code': 'BHR', 'new_name': 'Bahrain'},
    'CHN': {'new_code': 'CHN', 'new_name': "People's Republic of China"},
    'CIV': {'new_code': 'CIV', 'new_name': "Côte d'Ivoire"},
    'CZE': {'new_code': 'CZE', 'new_name': 'Czech Republic'},
    'EGY': {'new_code': 'EGY', 'new_name': 'Egypt'},
```

```
    'HKG': {'new_code': 'HKG', 'new_name': 'Hong Kong, China'},
    'KGZ': {'new_code': 'KGZ', 'new_name': 'Kyrgyzstan'},
    'MDA': {'new_code': 'MDA', 'new_name': 'Republic of Moldova'},
    'PRK': {'new_code': 'PRK', 'new_name': "Democratic People's Republic of Korea"},
    'SVK': {'new_code': 'SVK', 'new_name': 'Slovak Republic'},
    'KOR': {'new_code': 'KOR', 'new_name': 'Republic of Korea'},
    'TUR': {'new_code': 'TUR', 'new_name': 'Türkiye'},
    'GBR': {'new_code': 'GBR', 'new_name': 'Great Britain'},
    'USA': {'new_code': 'USA', 'new_name': 'United States of America'},
    'VEN': {'new_code': 'VEN', 'new_name': 'Venezuela'}
}

# iterate through each row, updating the country code and name according to the
corrections dictionary
for index, row in DimLocation.iterrows():
    if row['CountryCode'] in corrections:
        DimLocation.at[index, 'CountryCode'] = corrections[row['CountryCode']]['new_code']
        DimLocation.at[index, 'CountryName'] = corrections[row['CountryCode']]['new_name']

# print and check
print(DimLocation[['CountryCode', 'CountryName']])

# delete CountryName_olympic column
DimLocation.drop(['CountryName_olympic'], axis=1, inplace=True)

# need ContinentName from list-of-countries_areas-by-continent-2024
countries_data = pd.read_sql_table('list-of-countries_areas-by-continent-2024', engine)

# need ContinentName from list-of-countries_areas-by-continent-2024
countries_data = pd.read_sql_table('list-of-countries_areas-by-continent-2024', engine)


# check null value
print(DimLocation.isnull().sum())

# set Continent for Côte d'Ivoire
DimLocation.loc[DimLocation['CountryName'] == "Côte d'Ivoire", 'ContinentName'] = 'Africa'

# write into DW
DimLocation.to_sql('DimLocation', new_engine, if_exists='replace', index=False)
```

	CountryName text	CountryCode text	ContinentName text
1	Afghanistan	AFG	Asia
2	Argentina	ARG	South America
3	Armenia	ARM	Asia
4	Australia	AUS	Oceania
5	Austria	AUT	Europe
6	Azerbaijan	AZE	Asia
7	Belarus	BLR	Europe
8	Belgium	BEL	Europe
9	Brazil	BRA	South America
10	Bahrain	BHR	Asia
11	Burundi	BDI	Africa
12	Cameroon	CMR	Africa
13	Canada	CAN	North America
14	People's Republic of China	CHN	Asia
15	Colombia	COL	South America

> In processing the DimLocation dimension tables, country codes and country names were first extracted from the economic data tables and used as the basis for building the dimension tables. Next, country names were checked for consistency and inconsistencies were resolved by cross-referencing them with the Olympic data tables, especially for regions that may be represented differently in different data sources (e.g., `Hong Kong, China`). The inconsistencies found were corrected, including by updating country codes and country names. The country code in the data source of `Bahrain` should be `BHR`, I believe there is a mistake in the data source.
>
> In addition, records that could not be matched correctly because of incomplete data were removed, and continent names were set manually for some specific countries, for example, `Côte d'Ivoire`'s continent was set to Africa.

3.4.2 Year Dimension Table

```
# Year Dimension Table
# need to create a table with Year from 1980 to 2022
DimYear = pd.DataFrame({'Year': range(1980, 2023)})

# copy into DW
DimYear.to_sql('DimYear', new_engine, if_exists='replace', index=False)

import numpy as np
# need OlympicGame Year from olympic_hosts to present OlympicYearIndicator
hosts_data = pd.read_sql_table('olympic_hosts', engine)

# create YearID
```

```
DimYear['YearID'] = DimYear['Year']
DimYear = DimYear[['YearID', 'Year']]
DimYear['YearID'] = DimYear['YearID'].astype(str)

# merge with OlympicGame Year
DimYear = DimYear.merge(hosts_data[['Year']], on='Year', how='left', indicator=True)

# add a column OlympicYearIndicator
DimYear['OlympicYearIndicator'] = np.where(DimYear['_merge'] == 'both', 'Olympic Year',
'Non-Olympic Year')

# delete duplicate years
DimYear = DimYear.drop_duplicates(subset=['Year'], keep='first')

# delete _merge
DimYear.drop(columns=['_merge'], inplace=True)

DimYear['Year'] = DimYear['Year'].astype(str)

# write into DW
DimYear.to_sql('DimYear', new_engine, if_exists='replace', index=False)
```

	YearID text	Year text	OlympicYearIndicator text
1	1980	1980	Olympic Year
2	1981	1981	Non-Olympic Year
3	1982	1982	Non-Olympic Year
4	1983	1983	Non-Olympic Year
5	1984	1984	Olympic Year
6	1985	1985	Non-Olympic Year
7	1986	1986	Non-Olympic Year
8	1987	1987	Non-Olympic Year
9	1988	1988	Olympic Year
10	1989	1989	Non-Olympic Year
11	1990	1990	Non-Olympic Year
12	1991	1991	Non-Olympic Year
13	1992	1992	Olympic Year
14	1993	1993	Non-Olympic Year

> In processing the DimYear dimension table, I created a dataframe with year from 1980 to 2022. Pay attention to that `Year` should be a string, otherwise it would be counted as `measures` in Atoti.
>
> To distinguish between `Olympic` and `non-Olympic` years, year information was extracted from the Olympic host city data and merged with the Year dimension table. Through this step, a new column `OlympicYearIndicator` was added for each year, identifying whether the year was an Olympic year or not, using a conditional expression to assign a value based on whether the year existed in the Olympic

> host city data or not

3.4.3 Olympic Games Dimension Table

```
# Olympic Games Dimension Table
# need OlympicGame, HostCityName, OlympicSeason from olympic_hosts data
DimOlympicGames = pd.read_sql_table('olympic_hosts', engine)

# copy into DW
DimOlympicGames.to_sql('DimOlympicGames', new_engine, if_exists='replace', index=False)

# delete unnecessary columns
DimOlympicGames.drop(['Year'], axis=1, inplace=True)

# update CountryName and merge with CountryCode
DimOlympicGames['CountryName'] = DimOlympicGames['CountryName'].replace(name_mapping)
DimOlympicGames = DimOlympicGames.merge(DimLocation[['CountryName', 'CountryCode']],
on='CountryName', how='left')
print(DimOlympicGames)

# updata CountryCode for USSR and Yugoslavia
DimOlympicGames.loc[DimOlympicGames['HostCityName'] == 'Moscow', 'CountryCode'] = 'SUN'
DimOlympicGames.loc[DimOlympicGames['HostCityName'] == 'Sarajevo', 'CountryCode'] = 'YUG'

# delete unnecessary columns
DimOlympicGames.drop(['CountryName'], axis=1, inplace=True)

# write into DW
DimOlympicGames.to_sql('DimOlympicGames', new_engine, if_exists='replace', index=False)
```

	OlympicGame text	OlympicSeason text	HostCityName text	CountryCode text
1	beijing-2022	Winter	Beijing	CHN
2	tokyo-2020	Summer	Tokyo	JPN
3	pyeongchang-20...	Winter	PyeongChang	KOR
4	rio-2016	Summer	Rio	BRA
5	sochi-2014	Winter	Sochi	RUS
6	london-2012	Summer	London	GBR
7	vancouver-2010	Winter	Vancouver	CAN
8	beijing-2008	Summer	Beijing	CHN
9	turin-2006	Winter	Turin	ITA

> In building the Olympic Games dimension table, I extracted the Olympic Games name, host city name and Olympic season information from the Olympic host city data. To ensure consistency and accuracy between the Olympic Games data and the geolocation data, the country names were updated by using a pre-defined name mapping, and then merging them with the country codes in the Location dimension table by country name.

> In order to deal with historical country changes, for the Olympic Games held in Moscow and Sarajevo, their country codes were manually updated, labelled `Soviet Union (SUN)` and `Yugoslavia (YUG)`, respectively, in order to reflect the political-geographical reality of the time.

3.4.4 Event Dimension Table

```
# Event Dimension Table
# need DisciplineTitle, EventTitle and ParticipantType from olympic_medals
DimEvent = pd.read_sql_table('olympic_medals', engine)

# copy into DW
DimEvent.to_sql('DimEvent', new_engine, if_exists='replace', index=False)

# Extract year, then delete records before 1980
DimEvent['Year'] = DimEvent['OlympicGame'].str.extract('-(\d{4})').astype(int)
DimEvent = DimEvent[DimEvent['Year'] >= 1980]

# delete unnecessary columns
DimEvent.drop(['OlympicGame', 'AthleteGender', 'AthleteName', 'CountryName',
'CountryCode', 'MedalType', 'Year'], axis=1, inplace=True)

# de-duplication of events
DimEvent = DimEvent.drop_duplicates(subset=['DisciplineTitle', 'EventTitle',
'ParticipantType'])

# create unique EventID as PK
# reset index of DataFrame
DimEvent.reset_index(drop=True, inplace=True)
DimEvent.index += 1
DimEvent['EventID'] = DimEvent.index

cols = list(DimEvent.columns)
cols = [cols[-1]] + cols[:-1]
DimEvent = DimEvent[cols]

# write into DW
DimEvent.to_sql('DimEvent', new_engine, if_exists='replace', index=False)
```

	EventID bigint	DisciplineTitle text	EventTitle text	ParticipantType text
1	1	Freestyle Skiing	Men's Moguls	Athlete
2	2	Freestyle Skiing	Men's Freeski Halfpipe	Athlete
3	3	Freestyle Skiing	Men's Freeski Big Air	Athlete
4	4	Freestyle Skiing	Men's Ski Cross	Athlete
5	5	Freestyle Skiing	Women's Freeski Big Air	Athlete
6	6	Freestyle Skiing	Women's Moguls	Athlete
7	7	Freestyle Skiing	Women's Ski Cross	Athlete
8	8	Freestyle Skiing	Men's Aerials	Athlete
9	9	Freestyle Skiing	Women's Aerials	Athlete
10	10	Freestyle Skiing	Women's Freeski Halfpipe	Athlete
11	11	Freestyle Skiing	Women's Freeski Slopestyle	Athlete
12	12	Freestyle Skiing	Men's Freeski Slopestyle	Athlete
13	13	Short Track Speed Skating	Women's 500m	Athlete
14	14	Short Track Speed Skating	Men's 500m	Athlete

> I solved the problem of duplicate events by filtering out redundant entries based on the event name. Also, data before 1980 needed to be deleted. In addition, the `EventID` column was introduced as a primary key in order to create a unique identifier for each event.

3.4.5 Athlete Dimension Table

```
# Athlete Dimension Table
# need AthleteName and AthleteGender from olympic_medals
DimAthlete = pd.read_sql_table('olympic_medals', engine)

# copy into DW
DimAthlete.to_sql('DimAthlete', new_engine, if_exists='replace', index=False)

# Extract year, then delete records before 1980
DimAthlete['Year'] = DimAthlete['OlympicGame'].str.extract('-(\d{4})').astype(int)
DimAthlete = DimAthlete[DimAthlete['Year'] >= 1980]

# de-duplication of athletes
DimAthlete = DimAthlete.drop_duplicates(subset=['AthleteName', 'AthleteGender'])

# create unique AthleteID as PK
# reset index of DataFrame
DimAthlete.reset_index(drop=True, inplace=True)
DimAthlete.index += 1
DimAthlete['AthleteID'] = DimAthlete.index

cols = list(DimAthlete.columns)
cols = [cols[-1]] + cols[:-1]
DimAthlete = DimAthlete[cols]

# write into DW
```

```
DimAthlete.to_sql('DimAthlete', new_engine, if_exists='replace', index=False)
```

	AthleteID bigint	AthleteGender text	AthleteName text
1	1	Men	Mikael KINGSBURY
2	2	Men	Walter WALLBERG
3	3	Men	Ikuma HORISHIMA
4	4	Men	Nico PORTEOUS
5	5	Men	David WISE
6	6	Men	Alex FERREIRA
7	7	Men	Henrik HARLAUT
8	8	Men	Birk RUUD
9	9	Men	Colby STEVENSON
10	10	Men	Sergey RIDZIK
11	11	Men	Alex FIVA
12	12	Men	Ryan REGEZ
13	13	Women	Mathilde GREMAUD

> Athlete Dimension Table has a similar process as Event Dimension Table.

3.4.6 Economic Data Fact Table

```
# Fact Table
# Economic Data Fact Table
# copy into DW from Economic data
FactEconomic = pd.read_sql_table('Economic data', engine)
FactEconomic.to_sql('FactEconomic', new_engine, if_exists='replace', index=False)

# merge with CountryCode
FactEconomic = pd.merge(DimLocation, FactEconomic, on='CountryCode', how='left',
indicator=True)

FactEconomic = FactEconomic[FactEconomic['_merge'] == 'both']

# delete unnecessary columns
FactEconomic.drop(['CountryName_y', 'ContinentName', '_merge'], axis=1, inplace=True)
FactEconomic.rename(columns={'CountryName_x': 'CountryName'}, inplace=True)
FactEconomic.rename(columns={'Year': 'YearID'}, inplace=True)

FactEconomic['YearID'] = FactEconomic['YearID'].astype(int).astype(str)

# merge Global Population with DimLocation, get the right population for these countries
population_data = pd.read_sql_table('Global Population', engine)

pop_name_mapping = {
    'Bahamas, The': 'Bahamas',
```

```
    "China, People's Republic of": "People's Republic of China",
    'Hong Kong SAR':'Hong Kong, China',
    'Korea, Republic of': 'Republic of Korea',
    'North Macedonia ': 'North Macedonia',
    'Türkiye, Republic of': 'Türkiye'
}

population_data['CountryName'] = population_data['CountryName'].replace(pop_name_mapping)
FactEconomic = pd.merge(FactEconomic, population_data, on='CountryName', how='left',
indicator=True)

# delete unnecessary columns
FactEconomic.drop(['_merge'], axis=1, inplace=True)
FactEconomic.rename(columns={'2020': 'Population'}, inplace=True)

# merge life expectancy with FactEconomic using CountryCode
life_expectancy_data = pd.read_sql_table('life-expectancy', engine)

FactEconomic = pd.merge(FactEconomic, life_expectancy_data, on='CountryCode', how='left',
indicator=True)

# delete unnecessary columns
FactEconomic.drop(['CountryName_y','Year','_merge'], axis=1, inplace=True)
FactEconomic.rename(columns={'CountryName_x': 'CountryName'}, inplace=True)

# create unique EconomicFactID as PK
# reset index of DataFrame
FactEconomic.reset_index(drop=True, inplace=True)
FactEconomic.index += 1
FactEconomic['EconomicFactID'] = FactEconomic.index

cols = list(FactEconomic.columns)
cols = [cols[-1]] + cols[:-1]
FactEconomic = FactEconomic[cols]

# write into DW
FactEconomic.to_sql('FactEconomic', new_engine, if_exists='replace', index=False)
```

	EconomicFactID bigint	CountryName text	CountryCode text	YearID text	GDP double precision	HealthGDP double precision	Population double precision	LifeExpectancy double precision
1	1	Afghanistan	AFG	2020	516.87	15.53	32.94	62.58
2	2	Argentina	ARG	2020	8496.43	9.98	45.39	75.89
3	3	Armenia	ARM	2020	4505.87	12.24	2.96	72.17
4	4	Australia	AUS	2020	51722.07	10.65	25.63	84.32
5	5	Austria	AUT	2020	48809.23	11.47	8.9	81.5
6	6	Azerbaijan	AZE	2020	4229.91	4.61	10.07	66.87
7	7	Belarus	BLR	2020	6542.86	6.41	9.41	72.51
8	8	Belgium	BEL	2020	45517.9	11.06	11.52	80.79
9	9	Brazil	BRA	2020	6923.7	10.31	200.98	74.01
10	10	Bahrain	BHR	2020	23433.19	4.72	1.47	79.17
11	11	Burundi	BDI	2020	216.83	6.5	11.88	61.57
12	12	Cameroon	CMR	2020	1539.13	3.77	26.55	60.83
13	13	Canada	CAN	2020	43349.68	12.94	37.98	82.05
14	14	People's Republic of China	CHN	2020	10408.72	5.59	1412.12	78.08

> The first step involves merging this economic data with the `DimLocation` table on the `CountryName` field. This is performed to ensure that each economic record is matched with a corresponding country code from the `DimLocation` table. Another merge operation is conducted, population data aligned with the `DimLocation` table's country names using a mapping. The resulting DataFrame is merged into the `FactEconomic` DataFrame using the `CountryCode`.
>
> Additionally, life expectancy data is merged with the `FactEconomic` table using `CountryCode` again, ensuring that each economic record now also contains corresponding life expectancy.

3.4.7 Olympic Fact Table

```
# Olympic Fact Table
# get FactOlympic from olympic_medals
FactOlympic = pd.read_sql_table('olympic_medals', engine)

# copy into DW
FactOlympic.to_sql('FactOlympic', new_engine, if_exists='replace', index=False)

# create mapping
athlete_name_to_id = dict(zip(DimAthlete['AthleteName'], DimAthlete['AthleteID']))
event_title_to_id = dict(zip(DimEvent['EventTitle'], DimEvent['EventID']))

# replace content using mapping
FactOlympic['AthleteID'] = FactOlympic['AthleteName'].map(athlete_name_to_id)
FactOlympic['EventID'] = FactOlympic['EventTitle'].map(event_title_to_id)

# Extract year, then delete records before 1980
FactOlympic['YearID'] = FactOlympic['OlympicGame'].str.extract('-(\d{4})').astype(int)
FactOlympic = FactOlympic[FactOlympic['YearID'] >= 1980]
FactOlympic['YearID'] = FactOlympic['YearID'].astype(int).astype(str)

year_to_id = dict(zip(DimYear['Year'], DimYear['Year']))
FactOlympic['Year'] = FactOlympic['Year'].map(year_to_id)

# delete unnecessary columns
FactOlympic.drop(['DisciplineTitle', 'EventTitle', 'AthleteGender', 'ParticipantType',
'AthleteName', 'CountryName'], axis=1, inplace=True)
print(FactOlympic)

# create unique RecordID as PK
# reset index of DataFrame
FactOlympic.reset_index(drop=True, inplace=True)
FactOlympic.index += 1
FactOlympic['RecordID'] = FactOlympic.index

cols = list(FactOlympic.columns)
cols = [cols[-1]] + cols[:-1]
FactOlympic = FactOlympic[cols]

FactOlympic['AthleteID'] = FactOlympic['AthleteID'].astype(int)
FactOlympic['EventID'] = FactOlympic['EventID'].astype(int)
```

```
# write into DW
FactOlympic.to_sql('FactOlympic', new_engine, if_exists='replace', index=False)
```

	RecordID bigint	OlympicGame text	MedalType text	CountryCode text	AthleteID bigint	EventID bigint	YearID text
1	1	beijing-2022	SILVER	CAN	1	1	2022
2	2	beijing-2022	GOLD	SWE	2	1	2022
3	3	beijing-2022	BRONZE	JPN	3	1	2022
4	4	beijing-2022	GOLD	NZL	4	2	2022
5	5	beijing-2022	SILVER	USA	5	2	2022
6	6	beijing-2022	BRONZE	USA	6	2	2022
7	7	beijing-2022	BRONZE	SWE	7	3	2022
8	8	beijing-2022	GOLD	NOR	8	3	2022
9	9	beijing-2022	SILVER	USA	9	3	2022
10	10	beijing-2022	BRONZE	ROC	10	4	2022
11	11	beijing-2022	SILVER	SUI	11	4	2022
12	12	beijing-2022	GOLD	SUI	12	4	2022
13	13	beijing-2022	BRONZE	SUI	13	5	2022
14	14	beijing-2022	GOLD	CHN	14	5	2022

> To ensure the data references the correct dimensions, athlete names and event titles are converted to their respective IDs using predefined mappings. Next, I deleted the records prior to 1980 as they were no longer of value to my analysis. Also `Year` needed to map.
>
> During the data cleaning process, I found an incorrect country code `BRN` in the CountryCode column, which I corrected to the correct code `BHR` to ensure the accuracy of the data.
>
> Before finalizing the table, AthleteID and EventID columns are converted to integers for consistency.

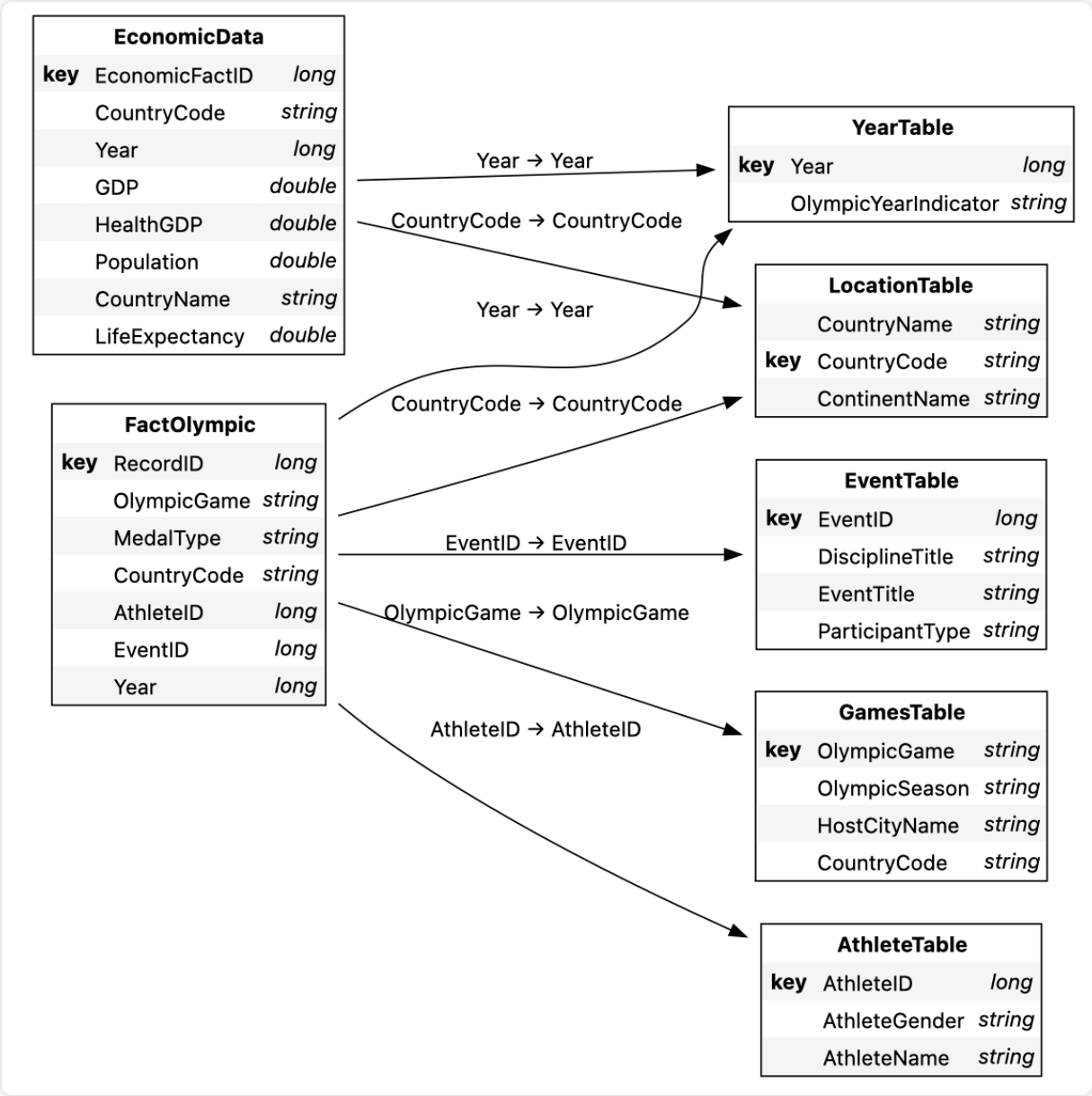
3.5 Cubes in Atoti

3.5.1 Schema visulization of Atoti

```
import atoti as tt
session = tt.Session()

# Connect to DB
jdbc_url = f"jdbc:postgresql://localhost:1225/OlympicDW?user=postgres&password=...."

# load date from DB
# fact table
olympic_fact_table = session.read_sql(
    "SELECT * FROM public.\"FactOlympic\"",
    keys=["RecordID"],
    table_name="FactOlympic",
    url=jdbc_url,
```

```
)


economic_data_table = session.read_sql(
    "SELECT * FROM public.\"FactEconomic\"",
    keys=["EconomicFactID"],
    table_name="EconomicData",
    url=jdbc_url,
)


# dimension tables
year_table = session.read_sql(
    "SELECT * FROM public.\"DimYear\"",
    keys=["Year"],
    table_name="YearTable",
    url=jdbc_url
)

location_table = session.read_sql(
    "SELECT * FROM public.\"DimLocation\"",
    keys=["CountryCode"],
    table_name="LocationTable",
    url=jdbc_url
)

games_table = session.read_sql(
    "SELECT * FROM public.\"DimOlympicGames\"",
    keys=["OlympicGame"],
    table_name="GamesTable",
    url=jdbc_url
)

event_table = session.read_sql(
    "SELECT * FROM public.\"DimEvent\"",
    keys=["EventID"],
    table_name="EventTable",
    url=jdbc_url
)

athlete_table = session.read_sql(
    "SELECT * FROM public.\"DimAthlete\"",
    keys=["AthleteID"],
    table_name="AthleteTable",
    url=jdbc_url
)

# implement a star schema
olympic_fact_table.join(location_table, (olympic_fact_table["CountryCode"] ==
location_table["CountryCode"]))
olympic_fact_table.join(games_table, (olympic_fact_table["OlympicGame"] ==
games_table["OlympicGame"]))
```

```
olympic_fact_table.join(event_table, (olympic_fact_table["EventID"] ==
event_table["EventID"]))
olympic_fact_table.join(athlete_table, (olympic_fact_table["AthleteID"] ==
athlete_table["AthleteID"]))
olympic_fact_table.join(year_table, (olympic_fact_table["Year"] == year_table["Year"]))

economic_data_table.join(location_table, (economic_data_table["CountryCode"] ==
location_table["CountryCode"]))
economic_data_table.join(year_table, (economic_data_table["Year"] == year_table["Year"]))

# visualise the schema
session.tables.schema
```

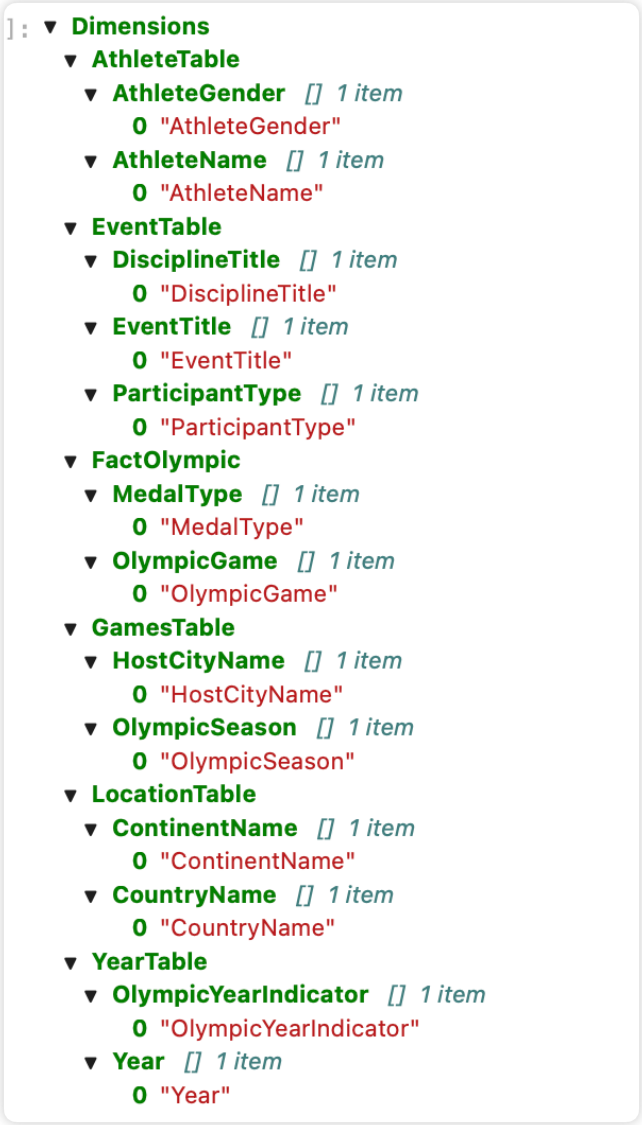


> A data cube is a data structure that allows data to be presented and analysed from multiple dimensions simultaneously.
> Roll-up aggregates data from lower levels to higher levels. For example, I can aggregate the number of gold medals for all European countries to the entire European continent level. This process reduces the granularity of the data by increasing the level of aggregation while providing a more macro view of

> the data. Drill-down is the opposite of roll-up and is a process of querying from a higher level to a lower level, which increases the granularity of the data. For example, drilling down from the continent level to the country level.
> In my data warehouse, cubes can be used to perform geographic analysis, athlete performance analysis and analysis of medals in different sports at the Olympics.

3.5.2 Olympic Fact Cube

```
# create a cube for olympic fact table
cube_olympic = session.create_cube(olympic_fact_table)
cube_olympic

# The cube has automatically created a hierarchy, level and measures
hierarchies_olympic, levels_olympic, measures_olympic = cube_olympic.hierarchies,
cube_olympic.levels, cube_olympic.measures

# check hierarchies_olympic
hierarchies_olympic

# delete unnecessary hierarchies
del hierarchies_olympic[('FactOlympic', 'CountryCode')]
del hierarchies_olympic[('FactOlympic', 'RecordID')]
del hierarchies_olympic[('FactOlympic', 'YearID')]
del hierarchies_olympic[('GamesTable', 'CountryCode')]

# check measures
measures_olympic

# delete useless measures
del measures_olympic["AthleteID.MEAN"]
del measures_olympic["AthleteID.SUM"]
del measures_olympic["EventID.MEAN"]
del measures_olympic["EventID.SUM"]
del measures_olympic["Year.MEAN"]
del measures_olympic["Year.SUM"]

measures_olympic["Medals_Count"] = measures_olympic["contributors.COUNT"]

del measures_olympic["contributors.COUNT"]
```

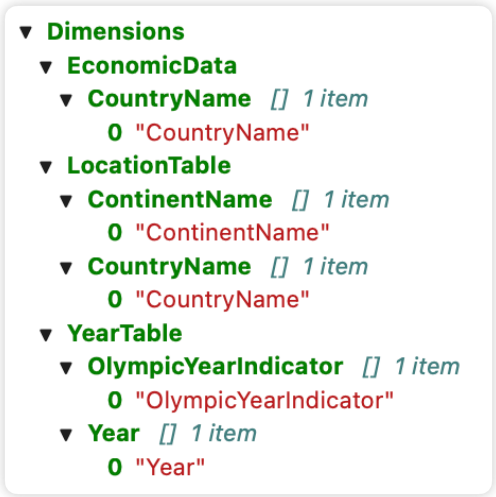


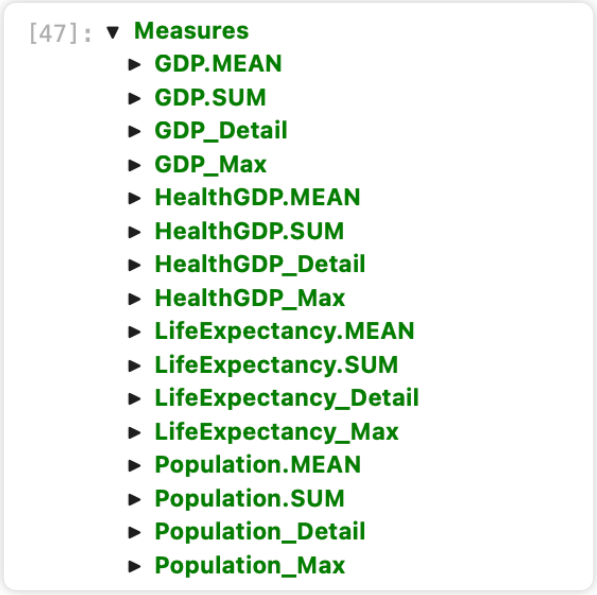
> In this fact table case, since a grain is a medal record, I only need to count the number of rows to get the number of medals without adding other measures.

3.5.3 Economic Data Fact Cube

```
# create a cube for economic data fact table
cube_economic = session.create_cube(economic_data_table)
cube_economic

# The cube has automatically created a hierarchy, level and measures
hierarchies_economic, levels_economic, measures_economic = cube_economic.hierarchies,
cube_economic.levels, cube_economic.measures

# check hierarchies_economic
hierarchies_economic

del hierarchies_economic[('EconomicData', 'CountryCode')]
```

```
del hierarchies_economic[('EconomicData', 'EconomicFactID')]
del hierarchies_economic[('EconomicData', 'YearID')]

# check measures
measures_economic

# delete useless measures
del measures_economic["contributors.COUNT"]

# add usefull measures

measures_economic["GDP_Detail"] = tt.agg.single_value(economic_data_table["GDP"])
measures_economic["HealthGDP_Detail"] =
tt.agg.single_value(economic_data_table["HealthGDP"])
measures_economic["LifeExpectancy_Detail"] =
tt.agg.single_value(economic_data_table["LifeExpectancy"])
measures_economic["Population_Detail"] =
tt.agg.single_value(economic_data_table["Population"])

measures_economic["GDP_Max"] = tt.agg.max(economic_data_table["GDP"])
measures_economic["HealthGDP_Max"] = tt.agg.max(economic_data_table["HealthGDP"])
measures_economic["LifeExpectancy_Max"] =
tt.agg.max(economic_data_table["LifeExpectancy"])
measures_economic["Population_Max"] = tt.agg.max(economic_data_table["Population"])
measures_economic
```





> There are many measures that can be calculated by Economic cube, and I've added many more here, but it is difficult to do SUM, MAX, etc., as there is only 2020 data. It is still the specific values that I mainly used in my analyses. But I hope to show my proficiency and use of the concept of measures through this process.

4 Clients' Queries and Visualization

For visualisation, as some of my queries spanned two cubes, it was difficult to use atoti visualisation (can't select attributes across cubes), so I only rendered bar charts etc. in some of the query's visualisation.

4.1 Client A: Sports Economics Researcher

4.1.1 Query 1

Relationship between GDP per capita of different countries in 2020 and the total number of gold medals at the 2020 Tokyo Olympics

```
gold_medals_2020 = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable","CountryName", "CountryName"]
    ],
    filter=(levels_olympic["YearTable","Year","Year"] == "2020") &
(levels_olympic["FactOlympic", "MedalType","MedalType"] == "GOLD")
)


gdp_per_capita_2020_top10 = cube_economic.query(
    measures_economic["GDP_Detail"],
    levels=[
        levels_economic["LocationTable","CountryName", "CountryName"]
    ]
).sort_values("GDP_Detail", ascending=False).head(10)

gold_medals_2020.join(gdp_per_capita_2020_top10, on="CountryName", how="inner")
```

CountryName	contributors.COUNT	GDP_Detail
Australia	17	51722.07
Ireland	2	85420.19
Norway	4	68340.02
Qatar	2	52315.66
Sweden	3	52837.9
United States of America	39	63528.63

> Not all countries with high GDP per capita get gold medals in the Olympics

4.1.2 Query2

Relationship between TOP 10 GDP per capita of different countries in 2020 and the total number of gold medals at all Winter Olympic Games

```
total_medals_winter_olympics = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable", "CountryName","CountryName"],
        levels_olympic["GamesTable","OlympicSeason", "OlympicSeason"]
    ],
    filter=(levels_olympic["GamesTable", "OlympicSeason", "OlympicSeason"] == "Winter")
)

total_medals_winter_olympics.join(gdp_per_capita_2020_top10, on="CountryName",
how="inner")
```

		contributors.COUNT	GDP_Detail
CountryName	OlympicSeason		
Australia	Winter	19	51722.07
Liechtenstein	Winter	8	165284.5
Luxembourg	Winter	2	117370.5
Norway	Winter	249	68340.02
Sweden	Winter	99	52837.9
United States of America	Winter	222	63528.63

> The countries with high GDP per capita are many European countries that can see that in comparison they are more prominent in the Winter Olympics, such as Norway and Sweden.

4.1.3 Query3

How many gold medals have countries in Asia 2020 with Health GDP per capita won at all Summer Olympics?

```
# Query3: How many gold medals have countries in Asia 2020 with Health GDP per capita won
at all Summer Olympics?
gold_medals_all_summer = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable", "CountryName","CountryName"],
        levels_olympic["GamesTable","OlympicSeason", "OlympicSeason"],
        levels_olympic["FactOlympic", "MedalType", "MedalType"]
    ],
    filter=(levels_olympic["GamesTable", "OlympicSeason","OlympicSeason"] == "Summer") &
           (levels_olympic["FactOlympic", "MedalType", "MedalType"] == "GOLD") &
           (levels_olympic["LocationTable", "ContinentName","ContinentName"] == "Asia")
    )
```

```
gdp_per_capita_asia_top3 = cube_economic.query(
    measures_economic["HealthGDP_Detail"],
    levels=[
        levels_economic["LocationTable","CountryName", "CountryName"]
    ],
    filter=(levels_olympic["LocationTable", "ContinentName","ContinentName"] == "Asia")
)

gold_medals_all_summer.join(gdp_per_capita_asia_top3, on="CountryName", how="inner")
```

			contributors.COUNT	HealthGDP_Detail
CountryName	OlympicSeason	MedalType		
Armenia	Summer	GOLD	2	12.24
Azerbaijan	Summer	GOLD	7	4.61
Bahrain	Summer	GOLD	2	4.72
Georgia	Summer	GOLD	10	7.6
India	Summer	GOLD	3	2.96
Israel	Summer	GOLD	3	8.32
Japan	Summer	GOLD	95	10.9
Jordan	Summer	GOLD	1	7.47
Kazakhstan	Summer	GOLD	15	3.79
People's Republic of China	Summer	GOLD	227	5.59
Qatar	Summer	GOLD	2	4.18
Republic of Korea	Summer	GOLD	85	8.36
Singapore	Summer	GOLD	1	6.05
Tajikistan	Summer	GOLD	1	8.18
Thailand	Summer	GOLD	10	4.36
Türkiye	Summer	GOLD	18	4.62
Uzbekistan	Summer	GOLD	10	6.75

4.1.4 Query4

How many medals have the top 1 countries in South America's 2022 GDP per capita won at the Beijing 2008 Olympics?

```
medals_2020 = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable","CountryName", "CountryName"]
    ],
    filter=(levels_olympic["YearTable","Year","Year"] == "2008")
```

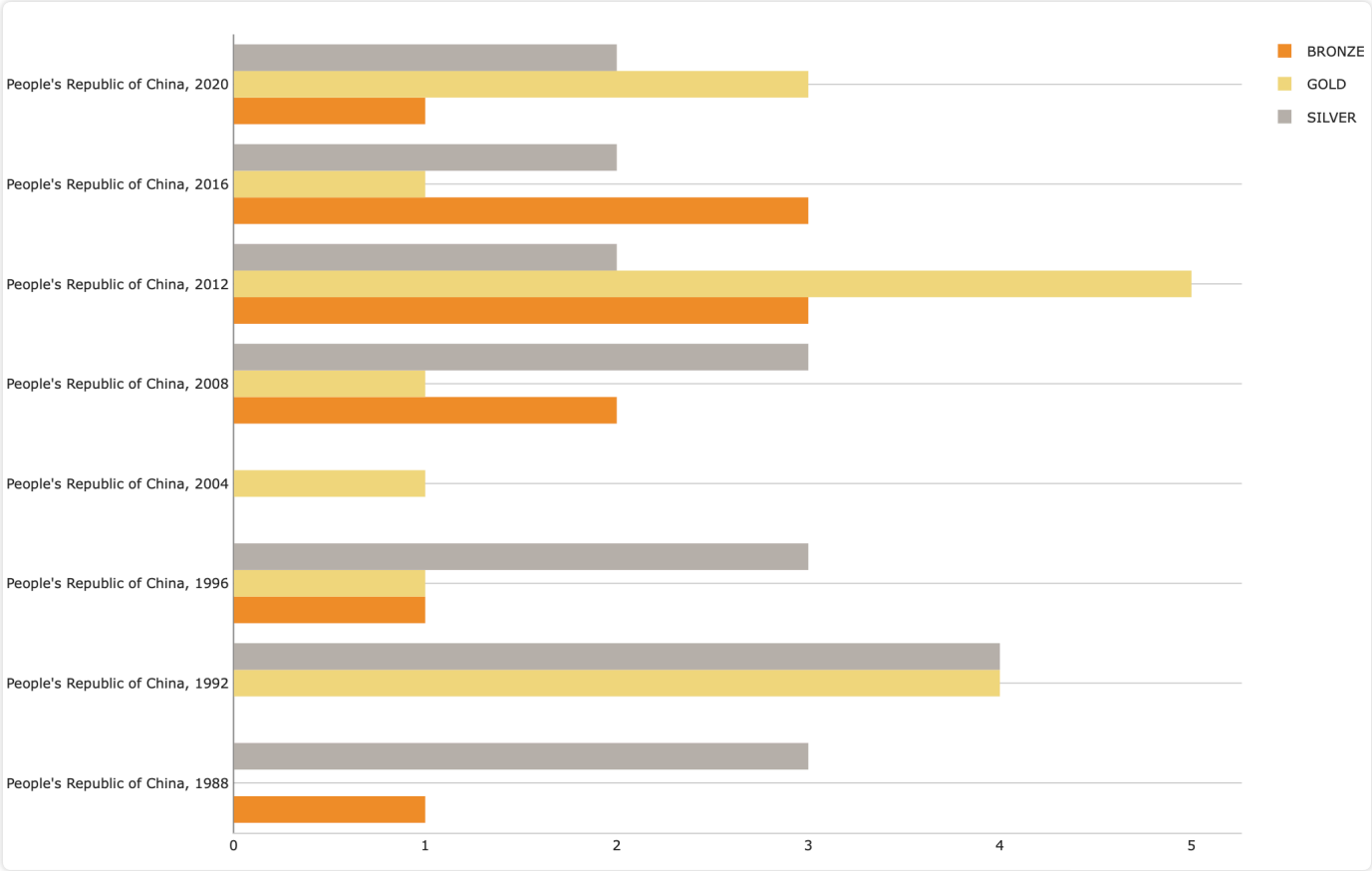
```
)

gdp_per_capita_2020_max = cube_economic.query(
    measures_economic["GDP_Detail"],
    levels=[
        levels_economic["LocationTable","CountryName", "CountryName"]
    ],
    filter=(levels_olympic["LocationTable", "ContinentName","ContinentName"] == "South
America")
).sort_values("GDP_Detail", ascending=False).head(1)

medals_2020.join(gdp_per_capita_2020_max, on="CountryName", how="inner")
```

CountryName	contributors.COUNT	GDP_Detail
Argentina	6	8496.43

4.1.5 Query5

What are the changes in the number of medals China has won at all the summer Olympic Games?

```
china_swim_medal = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable","CountryName", "CountryName"],
        levels_olympic["GamesTable","OlympicSeason", "OlympicSeason"],
        levels_olympic["YearTable","Year","Year"],
        levels_olympic["EventTable","DisciplineTitle","DisciplineTitle"],
        levels_olympic["FactOlympic", "MedalType", "MedalType"]
    ],
    filter=(levels_olympic["GamesTable", "OlympicSeason","OlympicSeason"] == "Summer") &

           (levels_olympic["LocationTable","CountryName", "CountryName"] == "People's
Republic of China")&
           (levels_olympic["EventTable","DisciplineTitle","DisciplineTitle"] ==
"Swimming")
)
```



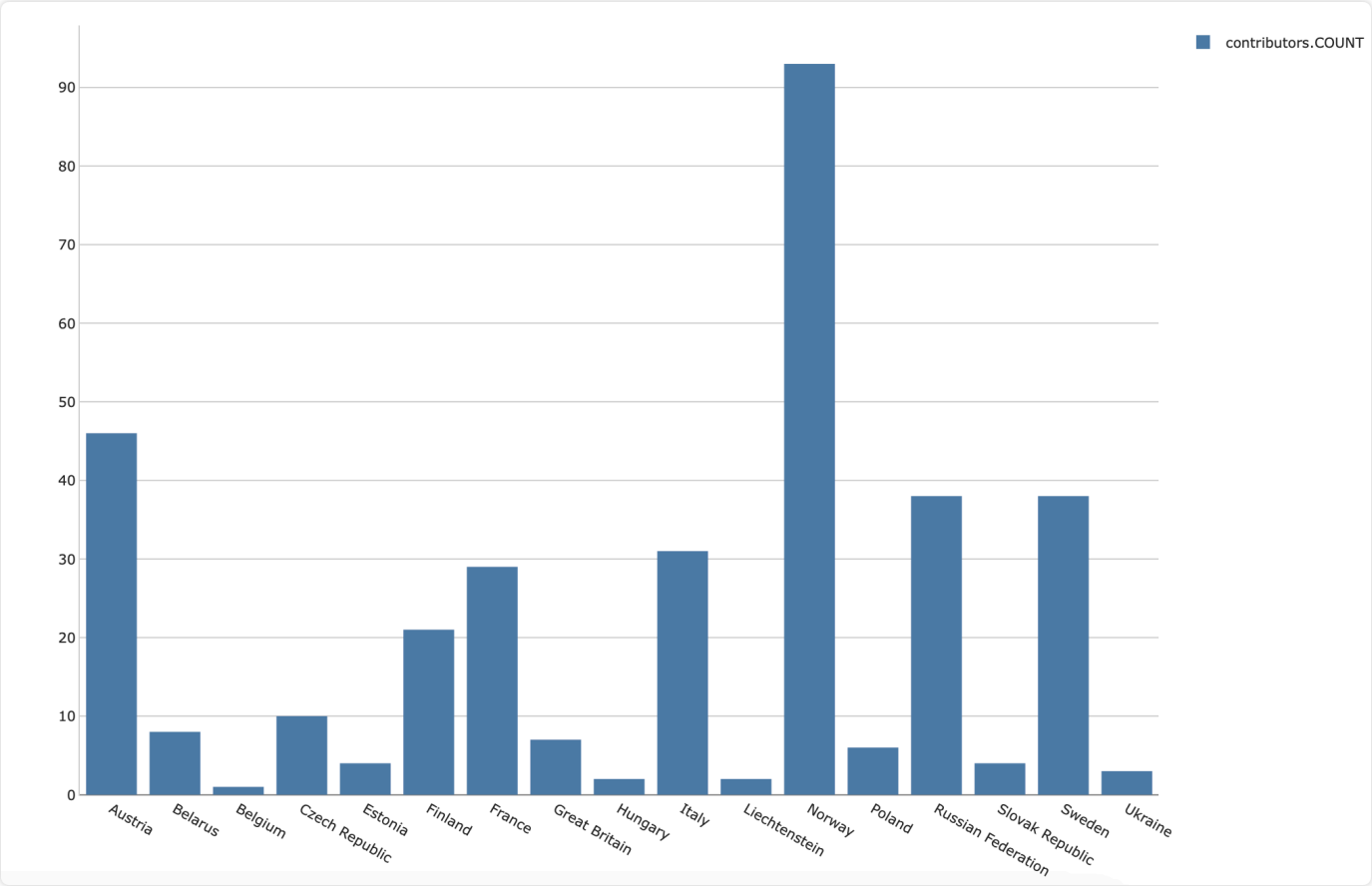
> The Chinese team's progress and performance in swimming is remarkable.

4.2 Client B: Sports Promotion Officer

4.2.1 Query1

The total number of gold medals won by European countries in all Winter Olympic Games

```
gold_medals_europe_winter = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable", "CountryName","CountryName"],
        levels_olympic["GamesTable","OlympicSeason", "OlympicSeason"],
        levels_olympic["FactOlympic", "MedalType", "MedalType"]
    ],
    filter=(levels_olympic["GamesTable", "OlympicSeason","OlympicSeason"] == "Winter") &
           (levels_olympic["FactOlympic", "MedalType", "MedalType"] == "GOLD") &
           (levels_olympic["LocationTable", "ContinentName","ContinentName"] == "Europe")

)
```



> There is no doubt that the Nordic countries have had an outstanding Winter Olympic performance.

4.2.2 Query2

Query the world's top 10 countries in terms of population and the number of medals they have won at previous Summer Olympics

```
medals = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable","CountryName", "CountryName"]
    ],
    filter=(levels_olympic["GamesTable", "OlympicSeason","OlympicSeason"] == "Summer")
)

population_2020_top10 = cube_economic.query(
    measures_economic["Population_Detail"],
    levels=[
        levels_economic["LocationTable","CountryName", "CountryName"]
    ]
).sort_values("Population_Detail", ascending=False).head(10)

medals.join(population_2020_top10, on="CountryName", how="inner")
```

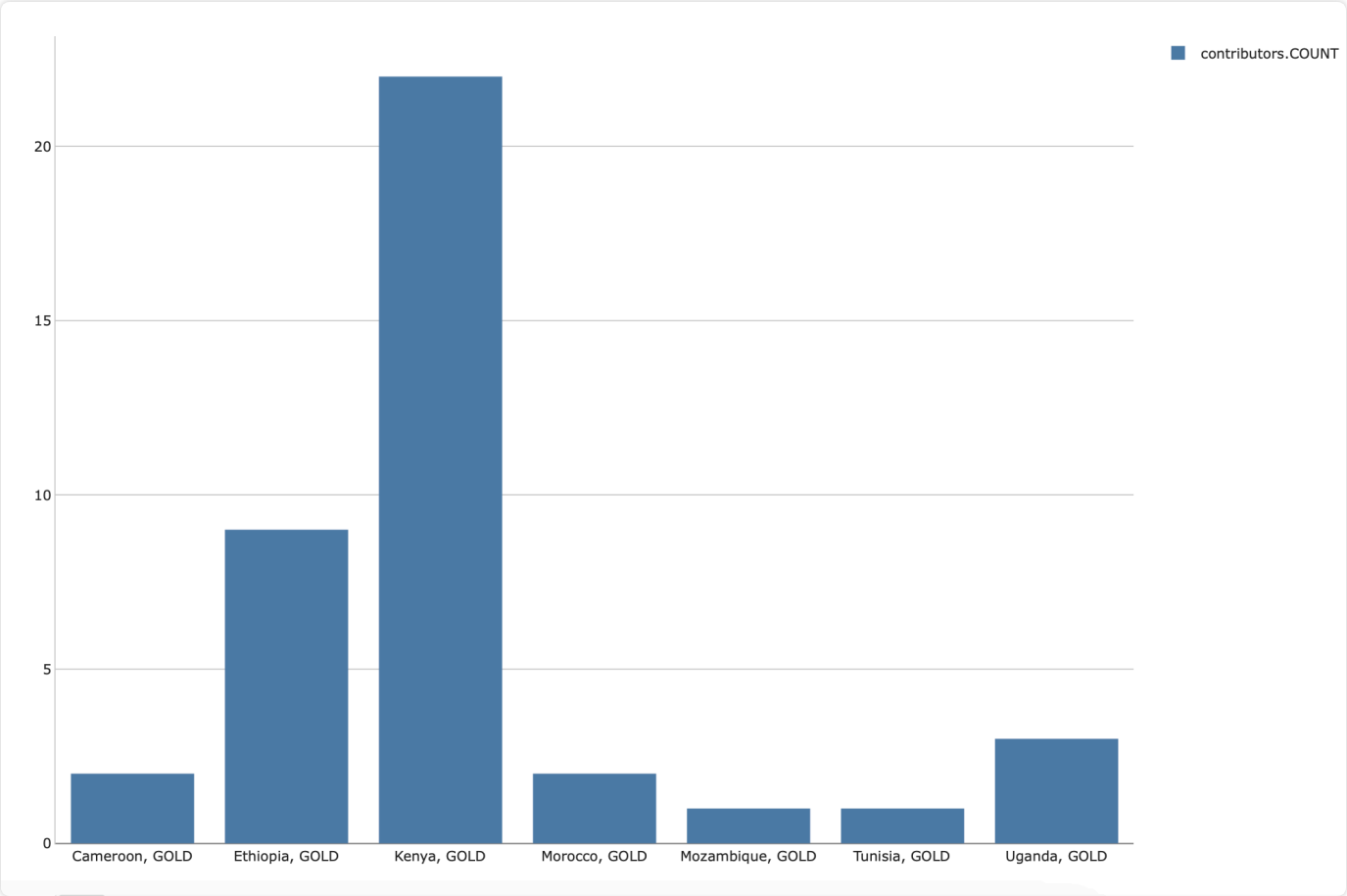
CountryName	contributors.COUNT	Population_Detail
Brazil	113	200.98
Egypt	21	100.6
Ethiopia	51	99.7
India	22	1396.39
Japan	284	125.85
Mexico	45	127.79
Pakistan	2	218.24
People's Republic of China	576	1412.12
Russian Federation	400	146.17
Türkiye	61	83.61

> The conclusion that more populous countries also win more medals at the Olympics is not unfounded, it is a positive correlation.

4.2.3 Query3

The total number of gold medals won by African countries in all Summer Olympic Games especially of Athletics discipline

```
gold_medals_africa_summer = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable", "CountryName","CountryName"],
        levels_olympic["GamesTable","OlympicSeason", "OlympicSeason"],
        levels_olympic["FactOlympic", "MedalType", "MedalType"]
    ],
    filter=(levels_olympic["GamesTable", "OlympicSeason","OlympicSeason"] == "Summer") &
           (levels_olympic["FactOlympic", "MedalType", "MedalType"] == "GOLD") &
           (levels_olympic["LocationTable", "ContinentName","ContinentName"] == "Africa")&
           (levels_olympic["EventTable","DisciplineTitle","DisciplineTitle"] ==
"Athletics")

)
gold_medals_africa_summer
```



> Kenya has won almost all the medals in Athletics (track and field) events.

4.2.4 Query4

What are the top 5 countries with the longest life expectancy per capita and the number of medals won at all summer Olympics

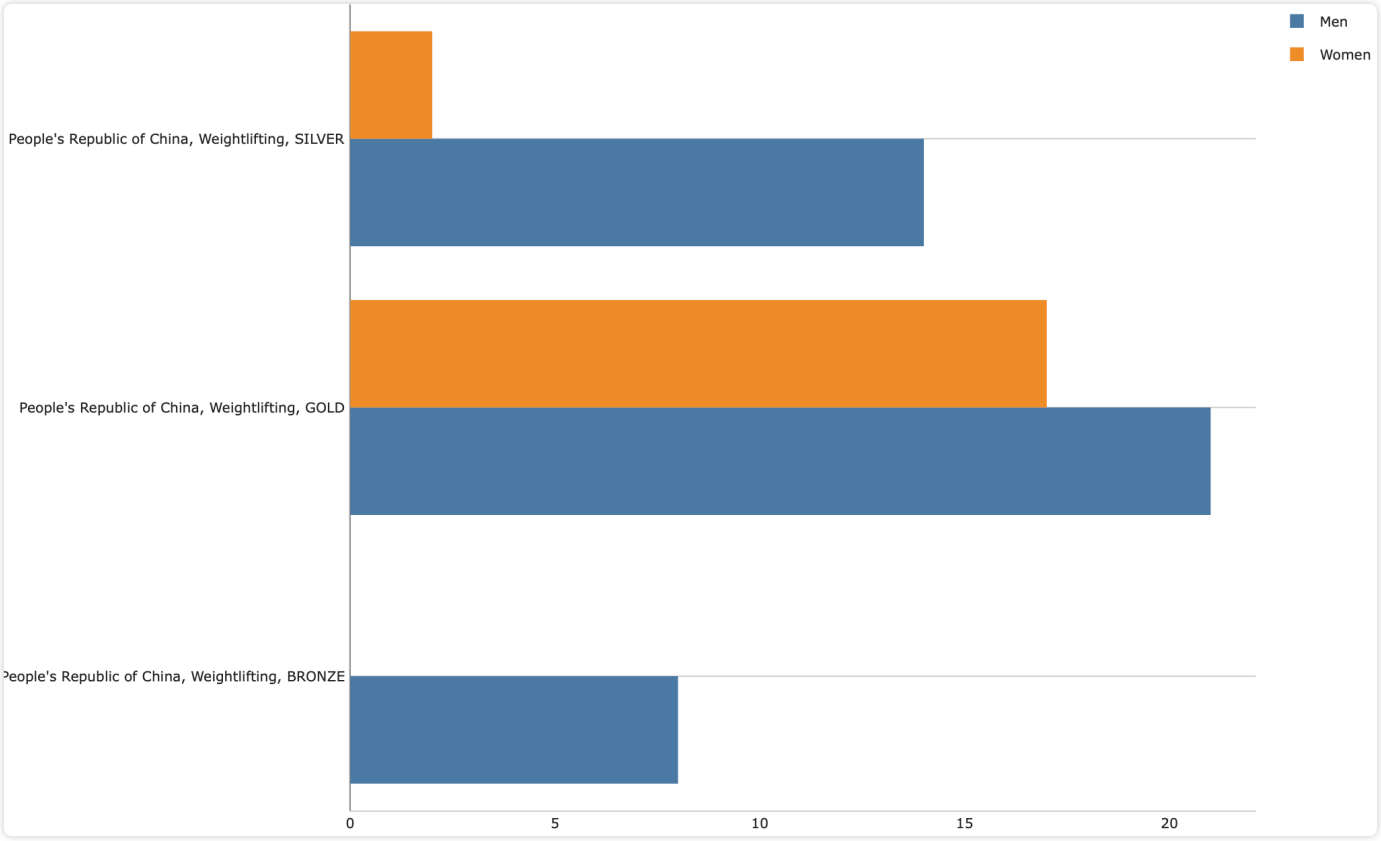
```
lifeexpectancy_2020_top5 = cube_economic.query(
    measures_economic["LifeExpectancy_Detail"],
    levels=[
        levels_economic["LocationTable","CountryName", "CountryName"]
    ]
).sort_values("LifeExpectancy_Detail", ascending=False).head(10)

medals.join(lifeexpectancy_2020_top5, on="CountryName", how="inner")
```

CountryName	contributors.COUNT	LifeExpectancy_Detail
Australia	330	84.32
Hong Kong, China	9	85.2
Iceland	3	82.58
Ireland	25	82.47
Japan	284	84.69
New Zealand	101	82.74
Norway	58	83.2
Republic of Korea	245	83.61
Singapore	4	82.86

4.2.5 Query5

Weightlifting has always been China's strongest sport, but who has had better results in previous Olympics for both men and women?

```
weightlift_medals_china = cube_olympic.query(
    measures_olympic["contributors.COUNT"],
    levels=[
        levels_olympic["LocationTable", "CountryName","CountryName"],
        levels_olympic["GamesTable","OlympicSeason", "OlympicSeason"],
                levels_olympic["EventTable", "DisciplineTitle","DisciplineTitle"],
        levels_olympic["FactOlympic", "MedalType", "MedalType"],
        levels_olympic["AthleteTable", "AthleteGender","AthleteGender"]
    ],
    filter=(levels_olympic["EventTable","DisciplineTitle","DisciplineTitle"] ==
"Weightlifting" ) &
             (levels_olympic["LocationTable","CountryName", "CountryName"] == "People's
Republic of China")

)
weightlift_medals_china
```



> China's male athletes have won more medals than females in weightlifting.

5 Rule Mining

```
# import necessary libraries, pandas, numpy and mlxtend
import numpy as np
import mlxtend
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules

# use pandas to connect and read SQL
import pandas as pd
from sqlalchemy import create_engine

connection_url = f"postgresql://postgres:....@localhost:1225/OlympicDW"
engine = create_engine(connection_url)

# load date from DB

DimLocation = pd.read_sql_table('DimLocation', engine)
DimYear = pd.read_sql_table('DimYear', engine)
DimOlympicGames = pd.read_sql_table('DimOlympicGames', engine)
DimEvent = pd.read_sql_table('DimEvent', engine)
DimAthlete = pd.read_sql_table('DimAthlete', engine)
FactOlympic = pd.read_sql_table('FactOlympic', engine)

# merge the table
FactOlympic = pd.merge(FactOlympic, DimLocation, left_on='CountryCode',
right_on='CountryCode', how='inner')
FactOlympic = pd.merge(FactOlympic, DimOlympicGames, left_on='OlympicGame',
right_on='OlympicGame', how='inner')
FactOlympic = pd.merge(FactOlympic, DimEvent, left_on='EventID', right_on='EventID',
how='inner')
FactOlympic = pd.merge(FactOlympic, DimAthlete, left_on='AthleteID', right_on='AthleteID',
how='inner')
FactOlympic = pd.merge(FactOlympic, DimYear, left_on='YearID', right_on='YearID',
how='inner')

FactOlympic = FactOlympic.rename(columns={
    'CountryCode_x': 'AthleteCountryCode',
    'CountryCode_y': 'HostCountryCode'
})
FactOlympic = FactOlympic.drop(columns=['EventID', 'AthleteID', 'YearID',
'OlympicYearIndicator', 'HostCountryCode'])

# check the data types in the dataframe
FactOlympic.dtypes

# check the columns of the dataframe
FactOlympic.columns

# check the number of rows and columns
```

```
FactOlympic.shape

# count total missing values at each column in the dataframe
FactOlympic.isna().sum()

columns_to_encode = ['CountryName', 'MedalType', 'DisciplineTitle']

# create dummy variables
olympic_encoded = pd.get_dummies(FactOlympic, columns=columns_to_encode)

print(olympic_encoded.head())
```

	RecordID	OlympicGame	MedalType	CountryCode_x	AthleteID	EventID	YearID	CountryName	ContinentName	OlympicSeason	HostCityName	CountryCode_y	DisciplineTitle	EventTitle
0	1	beijing-2022	SILVER	CAN	1	1	2022	Canada	North America	Winter	Beijing	CHN	Freestyle Skiing	Men's Moguls
1	2	beijing-2022	GOLD	SWE	2	1	2022	Sweden	Europe	Winter	Beijing	CHN	Freestyle Skiing	Men's Moguls
2	3	beijing-2022	BRONZE	JPN	3	1	2022	Japan	Asia	Winter	Beijing	CHN	Freestyle Skiing	Men's Moguls
3	19	beijing-2022	SILVER	CAN	19	7	2022	Canada	North America	Winter	Beijing	CHN	Freestyle Skiing	Women's Ski Cross
4	20	beijing-2022	GOLD	SWE	20	7	2022	Sweden	Europe	Winter	Beijing	CHN	Freestyle Skiing	Women's Ski Cross

```
# delete unnecessary columns
columns_to_drop = ['RecordID', 'AthleteCountryCode', 'ContinentName',
'OlympicSeason','HostCityName',  'EventTitle', 'ParticipantType', 'AthleteGender',
'AthleteName', 'Year','OlympicGame']
olympic_encoded.drop(columns=columns_to_drop, inplace=True)

# Apriori function
frequent_itemsets = apriori(olympic_encoded, min_support=0.02, use_colnames=True)
frequent_itemsets

# check the length of rules

frequent_itemsets['length'] = frequent_itemsets['itemsets'].apply(lambda x: len(x))

# #Assume the length is 2 and the min support is >= 0.03
filtered_itemsets = frequent_itemsets[(frequent_itemsets['length'] == 2) &
                                      (frequent_itemsets['support'] >= 0.03)]
filtered_itemsets
```

	support	itemsets	length
41	0.043167	(CountryName_United States of America, MedalTy...	2
42	0.051991	(MedalType_GOLD, CountryName_United States of ...	2
43	0.049964	(MedalType_SILVER, CountryName_United States o...	2
45	0.030289	(DisciplineTitle_Swimming, CountryName_United ...	2
46	0.041021	(DisciplineTitle_Athletics, MedalType_BRONZE)	2
49	0.030408	(DisciplineTitle_Swimming, MedalType_BRONZE)	2
51	0.038517	(MedalType_GOLD, DisciplineTitle_Athletics)	2
53	0.038755	(MedalType_SILVER, DisciplineTitle_Athletics)	2
54	0.030766	(MedalType_SILVER, DisciplineTitle_Swimming)	2

```
# assume the min confidence is 0.1
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.1)

# assume the min lift is 1
rules_lift = association_rules(frequent_itemsets, metric="lift", min_threshold=1)

# based on min confidence (=0.1)
# output antecedents, consequents, support, confidence and lift.
result = rules[['antecedents','consequents','support','confidence','lift']]
result
```

	antecedents	consequents	support	confidence	lift
0	(CountryName_France)	(MedalType_BRONZE)	0.020391	0.395833	1.083021
1	(CountryName_People's Republic of China)	(MedalType_BRONZE)	0.022061	0.285494	0.781126
2	(CountryName_People's Republic of China)	(MedalType_GOLD)	0.029692	0.384259	1.223386
3	(CountryName_People's Republic of China)	(MedalType_SILVER)	0.025519	0.330247	1.030685
4	(CountryName_Russian Federation)	(MedalType_BRONZE)	0.021226	0.354582	0.970154
5	(CountryName_Russian Federation)	(MedalType_GOLD)	0.020630	0.344622	1.097189
6	(CountryName_United States of America)	(MedalType_BRONZE)	0.043167	0.297453	0.813846
7	(MedalType_BRONZE)	(CountryName_United States of America)	0.043167	0.118108	0.813846
8	(MedalType_GOLD)	(CountryName_United States of America)	0.051991	0.165528	1.140604
9	(CountryName_United States of America)	(MedalType_GOLD)	0.051991	0.358258	1.140604
10	(MedalType_SILVER)	(CountryName_United States of America)	0.049964	0.155936	1.074510
11	(CountryName_United States of America)	(MedalType_SILVER)	0.049964	0.344289	1.074510
12	(DisciplineTitle_Athletics)	(CountryName_United States of America)	0.029335	0.247984	1.708786
13	(CountryName_United States of America)	(DisciplineTitle_Athletics)	0.029335	0.202136	1.708786
14	(DisciplineTitle_Swimming)	(CountryName_United States of America)	0.030289	0.335535	2.312076
15	(CountryName_United States of America)	(DisciplineTitle_Swimming)	0.030289	0.208710	2.312076
16	(DisciplineTitle_Athletics)	(MedalType_BRONZE)	0.041021	0.346774	0.948792
17	(MedalType_BRONZE)	(DisciplineTitle_Athletics)	0.041021	0.112235	0.948792

```
# find the rules whose confidence >= 0.3
new_result = result[result['confidence']>=0.3]

# assume k equals to 20
k = 20
top_k_rules = rules.nlargest(k, 'lift')
top_k_rules
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zhangs_metric
14	(DisciplineTitle_Swimming)	(CountryName_United States of America)	0.090269	0.145123	0.030289	0.335535	2.312076	0.017188	1.286565	0.623798
15	(CountryName_United States of America)	(DisciplineTitle_Swimming)	0.145123	0.090269	0.030289	0.208710	2.312076	0.017188	1.149680	0.663824
12	(DisciplineTitle_Athletics)	(CountryName_United States of America)	0.118292	0.145123	0.029335	0.247984	1.708786	0.012168	1.136780	0.470439
13	(CountryName_United States of America)	(DisciplineTitle_Athletics)	0.145123	0.118292	0.029335	0.202136	1.708786	0.012168	1.105086	0.485203
18	(DisciplineTitle_Boxing)	(MedalType_BRONZE)	0.049845	0.365490	0.024207	0.485646	1.328753	0.005989	1.233605	0.260394
19	(DisciplineTitle_Judo)	(MedalType_BRONZE)	0.051514	0.365490	0.023730	0.460648	1.260357	0.004902	1.176430	0.217794
2	(CountryName_People's Republic of China)	(MedalType_GOLD)	0.077272	0.314095	0.029692	0.384259	1.223386	0.005422	1.113951	0.197887
8	(MedalType_GOLD)	(CountryName_United States of America)	0.314095	0.145123	0.051991	0.165528	1.140604	0.006409	1.024452	0.179721
9	(CountryName_United States of America)	(MedalType_GOLD)	0.145123	0.314095	0.051991	0.358258	1.140604	0.006409	1.068818	0.144198
21	(DisciplineTitle_Wrestling)	(MedalType_BRONZE)	0.063439	0.365490	0.025876	0.407895	1.116021	0.002690	1.071617	0.111002
5	(CountryName_Russian Federation)	(MedalType_GOLD)	0.059862	0.314095	0.020630	0.344622	1.097189	0.001827	1.046579	0.094220
0	(CountryName_France)	(MedalType_BRONZE)	0.051514	0.365490	0.020391	0.395833	1.083021	0.001563	1.050223	0.080820
10	(MedalType_SILVER)	(CountryName_United States of America)	0.320415	0.145123	0.049964	0.155936	1.074510	0.003465	1.012811	0.102038
11	(CountryName_United States of America)	(MedalType_SILVER)	0.145123	0.320415	0.049964	0.344289	1.074510	0.003465	1.036410	0.081115
27	(DisciplineTitle_Swimming)	(MedalType_SILVER)	0.090269	0.320415	0.030766	0.340819	1.063680	0.001842	1.030954	0.065808
22	(MedalType_GOLD)	(DisciplineTitle_Athletics)	0.314095	0.118292	0.038517	0.122627	1.036645	0.001362	1.004941	0.051537
23	(DisciplineTitle_Athletics)	(MedalType_GOLD)	0.118292	0.314095	0.038517	0.325605	1.036645	0.001362	1.017067	0.040092
3	(CountryName_People's Republic of China)	(MedalType_SILVER)	0.077272	0.320415	0.025519	0.330247	1.030685	0.000760	1.014680	0.032265
24	(DisciplineTitle_Swimming)	(MedalType_GOLD)	0.090269	0.314095	0.029096	0.322325	1.026202	0.000743	1.012145	0.028067
25	(MedalType_SILVER)	(DisciplineTitle_Athletics)	0.320415	0.118292	0.038755	0.120953	1.022490	0.000852	1.003026	0.032365

5.1 Top k Rules

1. Antecedents: the conditional part of the rule, which, if it occurs, may lead to the occurrence of a later item.
2. Consequents: the result part of the rule, the items that usually appear when the antecedent appears.
3. Support: how often the antecedent and the consequent occur together.
4. Confidence: the conditional probability of the occurrence of the latter term given the occurrence of the former term. If the confidence is 1, it means that every time the antecedent occurs, the consequent always occurs.
5. Lift: the ratio of the frequency of occurrence of the antecedent and the consequent together to the product of the frequency of their independent occurrences. A lift greater than 1 means that there is a positive correlation between the antecedent and the consequent, i.e. they tend to occur together.

These correlations may point to the strategy of certain countries in their sports, or reveal their strength in particular Olympic events.

The rules indicate that certain countries have a higher probability of winning medals in specific sports events.

Typical associations that are more obvious and meaningful include:

- There is a strong correlation between swimming events and achieving results in the United States. This may reflect the strong strength and commitment of the United States in swimming.
- Athletics events are also highly correlated with achieving results in the United States.

- China has a high likelihood of winning gold medals whenever it competes, and a high percentage of silver medals, suggesting that in some areas the Chinese team has excelled.

5.2 Suggestions on Commerce

These tips may be useful for companies looking to partner with specific countries or promote products for specific sports.

- Sponsorship opportunities: consider sponsoring swimming and athletics events that are associated with the U.S., as these events are likely to attract significant attention and participation.
- Sports equipment: Design and market specialised sports equipment for countries that excel in swimming and athletics as a way to meet high performance needs.
- Sports training: create or offer courses dedicated to high-level swimming and athletics training, particularly in the United States.
- Marketing campaigns: marketing firms can design precision marketing campaigns that target specific sports and award-winning tendencies. For example, if data shows that China has a high likelihood of winning a gold medal at the Olympics, relevant adverts could be placed during the event or on sports channels.

6 Is Data Cube an Outdated Technology?

The concept of the Database Cube or Data Cube originated in the 1990s (Gray et al., 1997) when data warehousing and business intelligence technologies began to emerge (Cody, Kreulen, Krishna, & Spangler, 2002). They are primarily used to support On-line Analytical Processing (OLAP), to consolidate and bring together business data, to gain insight into the flow of data, and to apply analytical tools and techniques to make sense of the information in the data and make decisions to improve business efficiency. The various sources of relevant business data are known as Operational Data Stores (ODS). Data is extracted, transformed and loaded (ETL) from the ODS system to the data mart. In the data mart, the data is modelled as OLAP cubes (multi-dimensional models) to support flexible drill-down and roll-up analysis (Gangadharan & Swami, 2004).

However, database cubes are more complex and costly. the CUBE BY operation is resource-intensive and produces very large result sets, especially when the number of attributes and tuples in the original relation is high. Even with the availability of inexpensive high-capacity memory chips, it is challenging to store the entire data cube of a large relation in memory (Wang et al., 2002). Typically, data points in a data warehouse are dynamic, while dimensions are mostly static. However, dimensions often require updates. When such changes occur, the materialized aggregate views in OLAP systems need to be efficiently updated (Mumick, Quass, & Mumick, 1997). Data cubes typically contain a large number of pre-calculated aggregates, such as sums, averages, counts, etc. However, this precomputation is based on a static snapshot of an existing dataset. When new data is written, it cannot be calculated instantly. This is because data cubes are usually designed to optimise read operations for specific queries and reports, rather than frequent write operations. Therefore, it is also not suitable for frequently updated environments.

As cloud computing and big data technologies evolve, many of the aforementioned disadvantages are being overcome by a new generation of data warehousing services such as Amazon's Redshift and Google's BigQuery. These services offer greater flexibility, scalability, and better support for semi-structured and unstructured data.

Take Amazon Redshift as an example, it is a fast, fully managed, petabyte-scale data warehouse solution that makes it simple and cost-effective to efficiently analyse massive amounts of data using existing business intelligence tools. It provides a modern, columnar, horizontally scalable architecture that allows users to build data warehouses using EC2 instances and the database engine of their choice, as well as local or network-attached storage.

Amazon Redshift's data is loaded using a modified version of the PostgreSQL COPY command, and can be loaded directly from Amazon S3, Amazon DynamoDB, Amazon EMR, or over any SSH connection. In addition to the core Amazon Redshift software itself, several AWS services such as Amazon Elastic Compute Cloud (EC2) for instances, and Amazon S3 for backups, can greatly speed up Amazon Redshift development (Gupta et al., 2015).

6.1 Citiation

Gangadharan, G. R., & Swami, S. N. (2004, June). Business intelligence systems: design and implementation strategies. In *26th International Conference on Information Technology Interfaces, 2004.* (pp. 139-144). IEEE.

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., ... & Pirahesh, H. (1997). Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, *1*, 29-53.

Gupta, A., Agarwal, D., Tan, D., Kulesza, J., Pathak, R., Stefani, S., & Srinivasan, V. (2015, May). Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (pp. 1917-1923).

Mumick, I. S., Quass, D., & Mumick, B. S. (1997). Maintenance of data cubes and summary tables in a warehouse. *ACM Sigmod Record*, *26*(2), 100-111.

Wang, W., Feng, J., Lu, H., & Yu, J. X. (2002, February). Condensed cube: An effective approach to reducing data cube size. In *Proceedings 18th International Conference on Data Engineering* (pp. 155-165). IEEE.

W. F. Cody, J. T. Kreulen, V. Krishna and W. S. Spangler, "The Integration of Business Intelligence and Knowledge Management", *IBM Systems Journal*, vol. 41, no. 4, 2002.